ON ACCELERATING THE NONSYMMETRIC EIGENVALUE PROBLEM IN MULTICORE ARCHITECTURES

by

Matthew W. Nabity

M.S., University of Colorado Boulder, 2003

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Applied Mathematics

2013

This thesis for the Doctor of Philosophy degree by

Matthew W. Nabity

has been approved

by

Julien Langou, Advisor

Karen Braman

Lynn Bennethum

Jack Dongarra

Jan Mandel

July 12, 2013

Nabity, Matthew W. (Ph.D., Applied Mathematics)

On Accelerating the Nonsymmetric Eigenvalue Problem in Multicore Architectures

Thesis directed by Associate Professor Julien Langou

ABSTRACT

Scientific computing relies on state of the art algorithms in software libraries and packages. Achieving high performance in today's computing landscape is a challenging task as modern computer architectures in High-Performance Computing (HPC) are increasingly incorporating multicore processors and special purpose hardware, such as graphics processing units (GPUs). The emergence of this heterogeneous environment necessitates the development of algorithms and software packages that can fully exploit this structure. Maximizing the amount of parallelism within an algorithm is an essential step towards high performance.

One of the fundamental computations in numerical linear algebra is the computation of eigenvalues and eigenvectors. Matrix eigenvalue problems can be found in many applications. The algorithm of choice depends on the nature of the matrix. For sparse matrices, there are several viable approaches, but, for dense nonsymmetric matrices, one algorithm remains the method of choice. This is the QR algorithm. Unfortunately, on modern computer platforms, there are limitations to this approach in terms of parallelism and data movement. Current implementations do not scale well or do not take full advantage of the heterogeneous computing environment.

The goal of this work is to examine nonsymmetric matrix eigenvalue problems in the context of HPC. In Chapter 2, we examine tile algorithms and the implementation of block Arnoldi expansion in the context of multicore. Pseudocodes and implementation details are provided along with performance results.

In Chapter 3, we examine various algorithms in the context of computing a par-

tial Schur factorization for nonsymmetric matrices. We examine several iterative approaches and present implementations of specific methods. The methods studied include a block version of explicitly restarted Arnoldi with deflation, a block extension of Stewart's Krylov-Schur method, and a block version of Jacobi-Davidson. We present a new formulation of block Krylov-Schur that is robust and achieves improved performance for eigenvalue problems with sparse matrices. We experiment with dense matrices as well.

We expand on our work and present a C code for our block Krylov-Schur approach using LAPACK routines in Chapter 4. The code is robust and represents a first step towards an optimized version. Our code can use any desired block size and compute any number of desired eigenvalues. Detailed pseudocodes are provided along with a theoretical analysis.

The form and content of this abstract are approved. I recommend its publication.

Approved: Julien Langou

DEDICATION

To Blake and Breeann

ACKNOWLEDGMENT

First I want to thank my advisor Julien Langou for everything. His encouragement and guidance over the years has made this work possible. I want to thank my committee members for their patience and support; Professor Lynn Bennethum for guiding this process by being chair, Professor Karen Braman for the discussions over the years and for being a friendly face at conferences, Professor Jack Dongarra for the summer research opportunities at the Innovative Computing Laboratory (ICL) at the University of Tennessee Knoxville (UTK), and Professor Jan Mandel for pushing me as a student and researcher.

The work in this thesis was partially supported by the National Science Foundation Grants GK-12-0742434, NSF-CCF-1054864, and by the Bateman family in the form of the Lynn Bateman Memorial Fellowship.

This research benefited greatly by access to computing resources at the Center for Computational Mathematics (CCM) at the University of Colorado Denver and the ICL at UTK. The "colibri" cluster, NSF-CNS-0958354, was used for some of the experimental results presented in this work.

The journey was certainly enriched by the friendship of my fellow graduate students both at the University of Colorado Boulder and here in the Mathematics Department at the University of Colorado Denver.

Finally, I would not be here without the support of my family. I thank my parents for supporting me as I followed my own path. I owe a great deal to my brother Paul for constantly pushing his older brother to keep up with him. Lastly, finishing this work would not have been possible without the constant support of my Breeann.

TABLE OF CONTENTS

FIGURES

Figure

TABLES

Table

1. Introduction

This work is about computing with matrices, primarily those with complex entries. The field of complex numbers is denoted by $\mathbb{C}$, so the set of $n \times 1$ matrices, usually called column vectors, is denoted by $\mathbb{C}^n$ and the set of $m \times n$ matrices is denoted by $\mathbb{C}^{m\times n}$. Our main focus is on properties of square matrices, that is, matrices in $\mathbb{C}^{n\times n}$. Specifically, we are concerned with algorithms to compute eigenvalues and corresponding invariant subspaces, or eigenvectors, of square matrices. There are several types of matrices whose structure provides desirable properties one may exploit when formulating algorithms. These properties may lead to attractive numerical properties, such as stability, or attractive computational properties, such as efficient storage. Much has been done to formulate algorithms for the computation of eigenvalues and invariant subspaces for Hermitian matrices and symmetric matrices, see [63] and [71] for details. Many other structures that induce particular eigenvalue properties, such as block cyclic matrices, Hamiltonian matrices, orthogonal matrices, symplectic matrices and others have been studied extensively in [43, 70]. Our computational setting is problems involving nonsymmetric matrices which have no such underlying structure. These matrices may range from being extremely sparse, that is primarily consisting of zeros, or dense (i.e., not sparse). Eigenvalue problems with nonsymmetric matrices show up in many branches of the sciences and numerous collections of such matrices from real applications are maintained in efforts such as the Matrix Market [12] and the University of Florida Sparse Matrix Collection [22]. Example applications in the NEP Collection in Matrix Market include computational fluid dynamics, several branches of engineering, quantum chemistry, and hydrodynamics.

We begin with an introduction to the computing environment, a review of the theoretical foundations from linear algebra, and key algorithmic components. Most of the information presented in this section is needed introductory material for the

nonsymmetric eigenvalue problem (NEP) and subsequent sections, but it also serves as an overview of the state of the art with respect to numerical methods for the standard eigenvalue problem.

1.1 The Computing Environment

State of the art algorithms are the foundation of software libraries and packages that strive to achieve optimal performance in today's computing landscape. Modern computer architectures in High-Performance Computing (HPC) are increasingly incorporating multicore processors and special purpose hardware, such as graphics processing units (GPUs). The change to this multicore environment necessitates the development of algorithms and software packages that can take full advantage of this computational setting. One major challenge is maximizing the amount of parallelism within an algorithm, but there are many other issues related to designing effective algorithms which are nicely surveyed in [5].

Our work will make use of several standards in computing. The BLAS (Basic Linear Algebra Subprograms) library is used to perform essential operations [11]. Operations in numerical linear algebra are divided into three levels of functionality based on the type of data involved and the cost of the associated operation. Level 1 BLAS operations involve only vectors, Level 2 BLAS operations involve both matrices and vectors, and Level 3 BLAS operations involve only matrices. Level 3 BLAS operations are the preferred type of operation for optimal performance. Optimized BLAS libraries are often provided for specific architectures. The BLAS library can be multithreaded to make use of the multicore environment.

LAPACK (Linear Algebra PACKage) is a library based on BLAS that performs higher level operations common to numerical linear algebra [2]. Routines in LAPACK are coded with a specific naming structure where the first letter indicates the matrix data type, the next two indicate the type of matrix, and the last three indicate the computation performed by the program. We will refer to specific routines

generically using a lower case x in place of the matrix data type, so xGEQRT, which computes a block QR factorization of a general matrix, may refer to any of the variants SGEQRT (real), DGEQRT (double), CGEQRT (complex), or ZGEQRT (double complex). Algorithms in LAPACK are formulated to incorporate Level 3 BLAS operations as much as possible. This is accomplished by organizing algorithms so that operations are done with panels, either blocks of columns or rows of a matrix, rather than single columns or rows. While this perspective provides algorithms rich in Level 3 BLAS operations, there are disadvantages in the context of multicore. Memory architecture, synchronizations, and limited fine granularity can diminish performance. ScaLAPACK (Scalable LAPACK) is a library that includes a subset of LAPACK routines redesigned for distributed memory MIMD (multiple instruction multiple data) parallel computers [10]. The LAPACK and ScaLAPACK libraries are considered the standard for high performance computations in dense linear algebra. Both libraries implement sequential algorithms that rely on parallel building blocks. There are considerable advantages to reformulating old algorithms and developing new ones to increase performance on multicore platforms as demonstrated in [19, 18, 53].

PLASMA (Parallel Linear Algebra Software for Multicore Architectures) is a recent development focusing on tile algorithms, see [19, 18, 53], with the goal of addressing the performance issues of LAPACK and ScaLAPACK on multicore machines [1]. PLASMA uses a different data layout subdividing a matrix into square tiles as demonstrated in Figure 1.1. Operations restricted to small tiles create fine grained parallelism and provide enough work to keep multiple cores busy. The current version of PLASMA, release 2.4.6, relies on runtime scheduling of parallel tasks. Hybrid environments are developing as well that combine both multicore and other special purpose architectures like GPUs. Projects such as MAGMA (Matrix Algebra on GPU and Multicore Architectures) aim to build next generation libraries that fully exploit heterogeneous architectures [65].

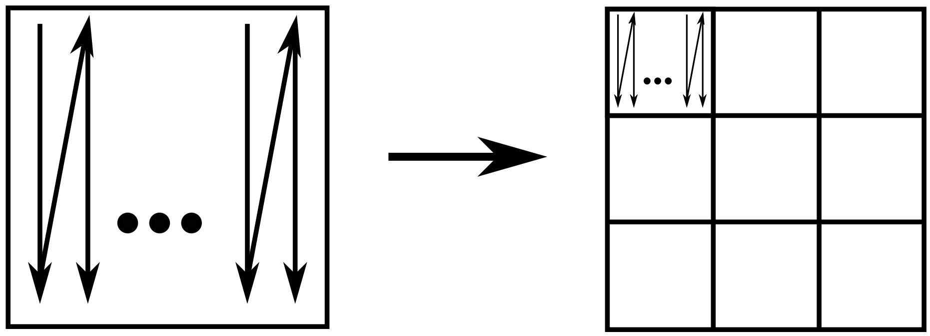


Figure 1.1: Data structure for tiled matrix

The multicore perspective necessitates the development of algorithms that can incorporate evolving computational approaches, such as tiled data structures that facilitate high performance. The work presented in ensuing chapters is focused on accelerating algorithms for the NEP in the context of multicore and hybrid architectures. We now turn to some basic theory of eigenvalues and an essential component of state-of-the-art algorithms.

1.2 The Standard Eigenvalue Problem

A nonzero vector $v \in \mathbb{C}^n$ is an eigenvector, or right eigenvector, of A if Av is a constant multiple of v. That is, there is a scalar $\lambda \in \mathbb{C}$ such that $Av = \lambda v$. The scalar λ is an eigenvalue associated with the right eigenvector v. Equivalently, an eigenvalue of a matrix $A \in \mathbb{C}^{n \times n}$ is a root of the characteristic polynomial

$$p_A(\lambda) \equiv |A - \lambda I| = 0,$$

where the vertical bars denote the determinant of the matrix $A - \lambda I$. A nonzero vector $y \in \mathbb{C}^n$ is a left eigenvector if it satisfies $y^H A = \lambda y^H$, where y^H is the conjugate transpose or Hermitian transpose of y. As our immediate focus is on the former, we will now refer to all right eigenvectors as simply eigenvectors unless the context is unclear. It is worth noting that some numerical methods utilize both left and right

eigenvectors. Often the pair (λ, v) is called an eigenpair and the set of all eigenvalues is the spectrum of A and is denoted by $\sigma(A)$. The eigenvalue associated with a given eigenvector is unique, but each eigenvalue has many eigenvectors associated with it as any nonzero multiple of v is also an eigenvector associated with λ. Spaces spanned by eigenvectors remain invariant under multiplication by A as

$$\text{span}\{Av\} \subseteq \text{span}\{\lambda v\} \subseteq \text{span}\{v\}.$$

This idea can be generalized to higher dimensions. A subspace $V \subset \mathbb{C}^n$ such that $AV \subseteq V$ is called a right invariant subspace of A. Again, there is a similar characterization of a left invariant subspace, but we will now work exclusively with the former and refer to them simply as invariant subspaces. An eigenvalue decomposition is a factorization of the form

$$A = X\Lambda X^{-1} \text{ or } AX = X\Lambda, \tag{1.1}$$

where Λ is a diagonal matrix with eigenvalues $\lambda_1, \ldots, \lambda_n$ on the diagonal and X is a nonsingular matrix with columns consisting of associated eigenvectors $v_1, \ldots, v_n$. A matrix is said to be nondefective if and only if it can be diagonalized, that is, it has an eigenvalue decomposition. Often nondefective matrices are called diagonalizable matrices. While a nice theoretical factorization, eigenvalue decompositions can be unstable calculations due to the conditioning of the eigenvector basis X. Fortunately there is a beautiful theoretical and computationally practical result, Schur's unitary triangularization theorem, which is a staple in linear algebra texts such as Horn and Johnson [35].

A central result to the computation of eigenvalues and eigenvectors is the Schur form of a matrix. For any matrix $A \in \mathbb{C}^{n\times n}$, there exists a unitary matrix Q such that

$$Q^H AQ = T \text{ or } AQ = QT, \tag{1.2}$$

where T is upper triangular. This factorization exists for any square matrix and the computation of this factorization is stable. More information on stability may be found in [32, 66] and we will discuss the stability of specific algorithms later, but for now it will suffice to hold stability as an attractive quality for algorithm design. The decomposition to Schur form is an eigenvalue revealing factorization as A and T are unitarily similar and the eigenvalues of A lie on the diagonal of T. The order of the eigenvalues on the diagonal may be organized by the appropriate choice of Q. It is often worthwhile to reorder the Schur factorization, for example to improve stability. A Schur decomposition is not unique as it depends on the order of the eigenvalues in T and does not account for eigenvalues with multiplicities greater than one. The associated eigenvectors, or invariant subspaces, may be computed from the Schur form. This is not an overly complicated computation, but there are some fundamental issues that can render computations unstable. Our primary focus is at the level of the Schur factorization and on algorithms designed to compute such a factorization. The matrix T may be complex when A is real. In this case, it may be advantageous to work with the real Schur form in which the matrix Q is now orthogonal and the matrix T is block upper triangular where the diagonal blocks are of order one or two depending on whether the eigenvalues are real or complex, respectively. In the case that A is Hermitian, $A^H = A$, the matrix T is a real diagonal matrix as the Schur form and the diagonalization of A are the same. As we will see, many algorithms compute a partial Schur form given by

$$AQ_k = Q_k T_k, \tag{1.3}$$

where $Q_k \in \mathbb{C}^{n \times k}$ with $k < n$ has orthonormal columns that span an invariant subspace and $T_k \in \mathbb{C}^{k \times k}$ is upper triangular or upper block triangular as described above.

1.3 Algorithms

In any approach, the computation of eigenvalues is necessarily an iterative process. This follows directly from the formulation of an eigenvalue problem as determining the roots of the characteristic polynomial, $p_A(\lambda)$, and Abel's well known theorem that there is no general algebraic solution to polynomials of degree five and higher. An extensive discussion of classic and more recent algorithmic approaches can be found in [28, 55, 63, 66, 70, 71]. In this section, we survey the major practical computational approach for computing a full Schur decomposition and the relevant state-of-the-art software.

The most essential algorithm to the NEP is the QR algorithm or QR iteration. First introduced by Francis [25, 26] and Kublanovskaya [44], the QR algorithm generates a sequence of orthogonally similar matrices that, under suitable assumptions, converges to a Schur form for a given matrix. The QR algorithm gets its name from the repeated computation of a QR factorization during the iteration. In this case, given an $n \times n$ matrix A, a QR factorization is a decomposition of the form $A = QR$ where Q is an $n \times n$ unitary matrix and R is an $n \times n$ upper triangular matrix. The most basic form of the QR algorithm computes a QR factorization, reverses the order of the factors and repeats until convergence. A more practical approach incorporates a preprocessing phase and shifts using eigenvalue estimates.

The first phase of a practical version of the QR algorithm requires the reduction of nonsymmetric A to Hessenberg form H. A matrix H is in upper Hessenberg form if $h_{ij} = 0$ whenever $i > j + 1$. Every matrix can be reduced to Hessenberg form by unitary similarity transformations. This reduction can be computed by using Householder reflectors as in the Arnoldi procedure further discussed in Chapter 2. An upper Hessenberg matrix H is in proper upper Hessenberg form, often called

irreducible, if $h_{j+1,j} \neq 0$ for $j = 1, \ldots, n-1$. If a matrix is not in proper upper Hessenberg form, it may be divided into two independent subproblems in which the matrices are proper upper Hessenberg. We will assume proper upper Hessenberg form and simply use the term Hessenberg from now on. Reduction to Hessenberg form is a cost saving measure aimed at reducing the number of flops in the iterative second phase as working with the unreduced matrix A is prohibitively expensive. Currently, reduction to Hessenberg form using block Householder reflectors is handled by calling LAPACK's xGEHRD or ScaLAPACK's PxGEHRD. We will discuss performance issues of this computation on multicore platforms in Chapter 3.

The second phase of a practical implementation works exclusively with the Hessenberg form. The modern implicit QR iteration takes on a much more complicated structure than its original formulation. We outline the main computational pieces of one step of the iteration.

Beginning with the Hessenberg matrix A, a select number of shifts or eigenvalue estimates are generated. Using these k shifts, $\rho_1, \ldots, \rho_k$, a QR factorization of the polynomial $p(A) = (A - \rho_1 I) \cdots (A - \rho_k I)$ is desired as this spectral transformation will speed up convergence. Explicit formulation of $p(A)$ is cost prohibitive, but computing $p(A)e_1$ where e_1 is the canonical basis vector is relatively inexpensive. Next unitary Q with first column $q_1 = \alpha p(A)e_1$, where $\alpha \in \mathbb{C}$, is constructed. Performing the similarity transformation

$$A \rightarrow Q^H A Q$$

disturbs the Hessenberg form as in Figure 1.2. In the next step of the iteration, the bulge is chased from the top left of the matrix to the bottom right corner returning the matrix to Hessenberg form as in Figure 1.3. The bulge is eliminated by applying similarity transformations, such as Householder reflectors, that introduce zeros in desired locations moving the bulge. This process of disturbing the Hessenberg form with eigenvalue estimates and chasing the bulge is continued until the desired Schur

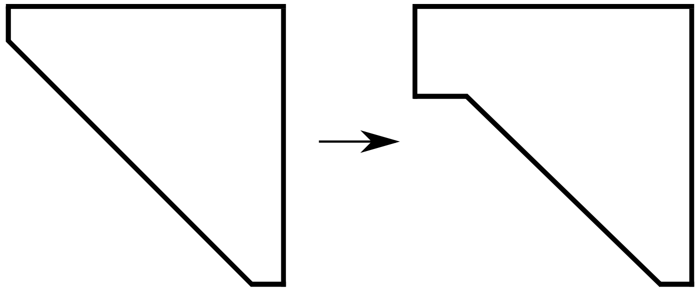


Figure 1.2: The Hessenberg structure disturbed

form is computed. There have been some major improvements to this process that form the foundation for state-of-the-art implementations of the implicit QR algorithm.

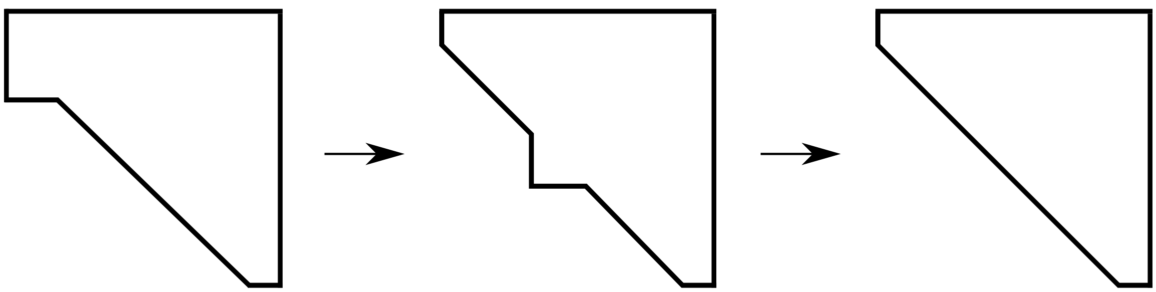


Figure 1.3: Chasing the bulge

After the introduction of the implicit shift approach in 1961, several implementations used single or double shifts creating multiple 1×1 or 2×2 bulges. These bulges were chased one column at a time using mainly Level 1 BLAS operations. In 1987, a multishift version of the QR algorithm was introduced by Bai and Demmel [8]. Here k simultaneous shifts were used creating a $k \times k$ bulge that was then chased p columns at a time. This was a significant step in the evolution of the QR algorithm as the restructured algorithm could be cast in more efficient Level 2 and Level 3 BLAS operations. Additionally, the k shifts were chosen to be the eigenvalues of the bottom

right $k \times k$ principal submatrix extending what had long been standard choices for shifts.

Though the multishift QR algorithm was able to be structured in efficient BLAS operations, the performance for a large number of shifts was lacking. Accurate shifts accelerate the convergence of the QR algorithm, but when a large number of shifts were used rounding errors developed degrading the shifts and slowing convergence. In 1992, Watkins [69] detailed this phenomenon of shift blurring. His analysis suggested that a small number of shifts be used maintaining a very small order bulge to avoid ill-conditioning and shift blurring. This proved to be an important idea that motivated a solution to using a large number of shifts and maintaining “well focused” shifts.

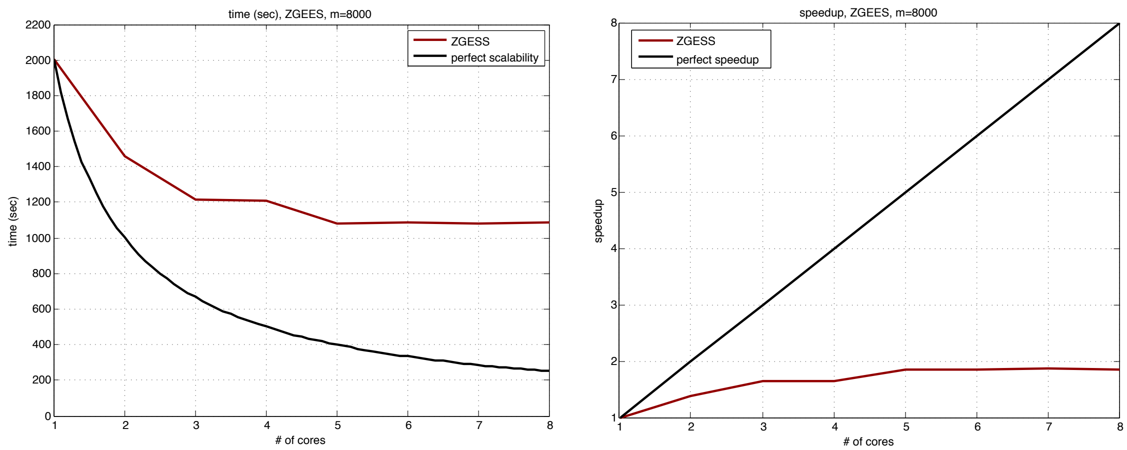
The seminal work by Braman, Byers and Mathias [15, 16] on multishift QR with aggressive early deflation (AED) introduced two new components that contributed greatly to the current success of the QR algorithm. Rather than k shifts and a single bulge as depicted in Figure 1.2, a chain of several tightly coupled bulges, each consisting of two shifts, is chased in the course of one iteration. This idea facilitated the use of Level 3 BLAS operations for most of the computational work and allowed the use of a larger number of shifts without the undesirable numerical degradation of shift blurring. Additionally, the use of AED located converged eigenvalues much faster than earlier deflation strategies which had changed little since the introduction of implicitly shifted QR. Together, these improvements reduced the number of iterations required by the QR algorithm greatly increasing overall performance. The LAPACK routine xHSEQR is the state-of-the-art serial implementation that computes the Schur form beginning with a Hessenberg matrix. The entire process, reduction to Hessenberg form and then Schur form, is performed by xGEES.

Another major improvement concerns the nontrivial task of parallelizing the QR algorithm. Parallel versions of the QR algorithm for distributed memory had been previously proposed, by Stewart [62], and work had been done on parallelizing the

multishift QR algorithm. The issue of shift blurring forced most efforts to focus on bulges of order 2 and scalability was still an issue. Issues pertaining to scalability of the standard double implicit shift QR algorithm were explored by Henry and van de Geijn [31]. In 1997, an approach using k shifts to form and chase $\frac{k}{2}$ bulges in parallel was presented by Henry, Watkins, and Dongarra [30] and subsequently added to ScaLAPACK. A novel approach based off of the multishift QR with AED was formulated by Granat, Kågström, and Kressner [29]. Here multi-window bulge chain chasing was formulated along with a parallelized version of AED. The software presented outperformed the existing ScaLAPACK implementation PxLAHQR.

Improving the QR algorithm continues to be a topic of interest. Recent work by Braman [14] investigated adjusting the AED strategy to find deflations in the middle of the matrix. Such a strategy could lead to a new divide and conquer formulation of the QR algorithm. Even more recent work by Karlsson and Kressner [39] examined optimal packing strategies for the chain of bulges in an effort to make effective use of Level 3 BLAS operations. A modified LAPACK implementation was presented and numerical results demonstrated the success of the approach. Optimally packed chains of bulges should aid the performance of parallel implementations of the QR algorithm as well.

Though the QR algorithm is the method of choice when computing a full Schur decomposition, there are some limitations to this approach in terms of parallelism and data movement. The current implementation of xGEES does not scale well. An important aspect of performance analysis is the study of how algorithm performance varies with problem size, the number of processors, and related parameters. Of particular importance is the scalability of an algorithm, or how effective it can use an increased number of processors. One approach at quantifying scalability is to determine how execution time varies with the number of available processors. To assess ZGEES, we recorded the execution time for computing the Schur factorization of an



(a) Timing

(b) Speed Up

Figure 1.4: Scalability of ZGEES

$8,000 \times 8,000$ matrix for up to 8 cores. The results along with a curve depicting perfect scalability can be seen in Figure 1.4a. We also illustrate the measured speed up and perfect speed up as the number of cores is increased from 1 to 8 in Figure 1.4b. As depicted in Figure 1.4, ZGEES does not scale well in the context of multicore. In addition to its performance, for the NEP, the only algorithm available in LAPACK for computing the Schur form is xGEES. Currently partial Schur forms for nonsymmetric matrices are unattainable in LAPACK.

In Table 1.1 we list the currently available computational approaches for the NEP available on various platforms. The methods listed in Table 1.1 are abbreviated as follows: Arnoldi based approaches are denoted by (A), implicitly restarted Arnoldi by (IRAM), Krylov-Schur based approaches by (KS), and Jacobi-Davidson methods by (JD). For packages based on Arnoldi, method (A), we include all formulations of Arnoldi such as the use of Chebyshev acceleration, preconditioning with Chebyshev polynomials, explicit restarts, and deflation, but not implicit restarts. Of the methods listed in Table 1.1, the only block Krylov-Schur implementation is part of the Anasazi package which is part of the Trilinos library. This implementation

Table 1.1: Available Iterative Methods for the NEP

Software	Routine	Method	Blocked	Language	Architecture
ARPACK	Various	IRAM	No	Fortran	Shared, Dist
SLEPc	EPSARNOLDI	A	No	Fortran, C	Shared, Dist
SLEPc	EPSKRYLOVSCHUR	KS	No	Fortran, C	Shared, Dist
SLEPc	EPSJD	JD	No	Fortran, C	Shared, Dist
Anasazi	BlockArnoldi	A	Yes	Fortran, C	Shared, Dist
Anasazi	BlockKrylovSchur	KS	Yes	Fortran, C	Shared, Dist
HSL 2013	EB13	A	Both	Fortran, C	Shared, Dist

uses two orthogonalization schemes, one proposed by Daniel, Gragg, Kaufman and Stewart (DGKS) [20] and a more recent offering by Stathopoulos and Wu [60] with the latter as the default setting.

The goal of this work is to attack the NEP in the context of HPC from a different approach. Our work concerns the computation of a partial Schur form via standard iterative techniques. To this end, in Chapter 2 we examine tile algorithms and the implementation of block Arnoldi expansion in the multicore context of PLASMA. The process constructs an orthonormal basis for a block Krylov space, and we extend existing algorithms with our tiled version. Pseudocodes and implementation details are provided along with performance results.

In Chapter 3, we examine various algorithms in the context of computing a partial Schur factorization for the nonsymmetric matrices. We examine several iterative approaches and present novel implementations of specific methods. The methods studied include a block version of explicitly restarted Arnoldi with deflation, a block extension of Stewart's Krylov-Schur method, and a block version of Jacobi-Davidson.

We compare our implementations to existing unblocked and blocked codes when available. All work is done in TATLAB and extensive numerical results are performed. This work motivates our algorithmic design choices in Chapter 4.

Finally, in Chapter 4 we present a detailed implementation of our block Krylov-Schur method using Householder reflectors. Our approach features a block algorithm, the ability to compute a full Schur decomposition, and the novel use of Householder reflectors consistent with the work in Chapter 3.

In this thesis we will use two distinct ways to parallelize our codes. In Section 2, we use the task-based scheduler QUARK from the University of Tennessee to parallelize our tile Arnoldi expansion. The tile Arnoldi expansion is written in term of sequential kernels, then dependencies between tasks are declared by labelling the variables as INPUT, INOUT, OUTPUT, finally, the QUARK scheduler unravels our code, figures out the task depedencies and exploits any parallelism present in our application. In Section 4, we obtain parallelism by calling multithreaded BLAS. (This is a similar parallelism model to the one in the LAPACK library.) In both cases, we relied on a third party to perform the parallelization per se. Both mechanisms are fairly high level and are indeed easy to use.

1.4 Contributions

Here we outline the specific contributions of this manuscript and associated work. In Chapter 2, we present our novel tiled implementation of block Arnoldi with Householder reflectors. The Arnoldi computation is an important component of both algorithms designed to solve linear systems and those used to compute eigenvalues. A great deal of time is spent in the Arnoldi component when working in either computational context and we present a marked improvement in performance with our tile approach. We managed to merge the orthogonalization with the matrix vector product. This has the potential to increase the performance of methods using block Arnoldi factorizations such as block GMRES and Morgan's GMRES-DR [51].

Additionally, any eigenvalue solver using block Arnoldi stands to benefit from this improvement.

The novel formulation of block Krylov-Schur with Householder in Chapter 3 also improves upon the state of the art. We present a new algorithm based on Householder reflectors rather than other orthogonalization schemes. Our robust formulation performs very well in the sparse case when computing partial Schur decompositions. We present numerical experiments in Matlab in Chapter 3 that suggest our approach is worth implementing in a compilable programming language.

In Chapter 4 we implement our block Krylov-Schur approach in a C code using LAPACK subroutines. The code is robust and supports any block sizes and can compute any number of desired eigenvalues. This code is the first step towards an optimized version that could be released to the scientific computing community.

2. Tiled Krylov Expansion on Multicore Architectures

In this chapter, we present joint work with Henricus Bouwmeester.

Many algorithms that aim to compute eigenvalues and invariant subspaces rely on Krylov sequences and Krylov subspaces. Additionally, many algorithms computing solutions to linear systems do as well. Here we consider the computation of an orthonormal basis for a Krylov subspace in the context of HPC. We are interested in an algorithm rich in BLAS Level 3 operations that achieves a high level of parallelism. We review some basic theory of Krylov subspaces and associated algorithms to motivate our current work. A wealth of information on Krylov subspaces and their connection to linear systems and eigenvalue problems may be found in [43, 55, 70].

If $A \in \mathbb{C}^{n\times n}$ is a matrix and $v \in \mathbb{C}^n$ is a nonzero vector, then the sequence $v, Av, A^2v, A^3v, \ldots$ is called a Krylov sequence. The subspace

$$\mathcal{K}_m(A, v) = \text{ span}\{v, Av, A^2v, \ldots, A^{m-1}v\}$$

is called the mth Krylov subspace associated with A and v and the matrix

$$K_m(A, v) = \left[v, Av, A^2v, \ldots, A^{m-1}v\right]$$

is called the mth Krylov matrix associated with A and v. Our current computational focus is on constructing a basis for the subspace $\mathcal{K}_m(A, v)$ as this Krylov subspace will play a pivotal role in many linear algebra computations, especially certain numerical methods for eigenvalue problems. Computing an explicit Krylov basis of the form $K_m(A, v)$ is not advisable. As m increases, under mild assumptions on the starting vector, the vectors $A^{m-1}v$ converge to the eigenvector associated with the largest eigenvalue in magnitude of A (provided th eigenvalue is simple). As m gets larger, the basis $[v, Av, A^2v, \ldots, A^{m-1}v]$ becomes extremely ill-conditioned and, consequently, much of the information in this basis is corrupted by roundoff errors as discussed by Kressner [43]. An elegant algorithm due to W. E. Arnoldi is one way to to compute a

basis for the Krylov space with better conditioning. The algorithm has a few variants which we explore in the next section.

2.1 The Arnoldi Method

In 1951, Walter E. Arnoldi [4], expanding Lanczos's work on eigenvalues of symmetric matrices, introduced an algorithm that reduced a general matrix to upper Hessenberg form. Let $v_1, v_2, \ldots, v_m$ be the result of sequentially orthonormalizing the Krylov sequence $v, Av, \ldots, A^{m-1}v$ and let $V_m = [v_1, v_2, \ldots, v_m]$. In matrix terms, Arnoldi's procedure generates a decomposition of the form

$$AV_m = V_m H_m + \beta_m v_{m+1} \mathbf{e}_m^T \tag{2.1}$$

where $H_m \in \mathbb{C}^{m\times m}$ is upper Hessenberg, β_m is a scalar, V_m has orthonormal columns, and $\mathbf{e}_m \in \mathbb{C}^m$ is the vector with one in the mth position and zeros everywhere else. This factorization is called an Arnoldi decomposition of order m, or simply an Arnoldi decomposition. We can represent this in matrix terms by

$$AV_m = V_{m+1}\widetilde{H}_{m+1} \tag{2.2}$$

where

$$\widetilde{H}_{m+1} = \begin{bmatrix} H_m \\ \beta_m \mathbf{e}_m^T \end{bmatrix}.$$

Arnoldi suggested that the matrix H_m may contain accurate approximations to the eigenvalues of A. We will revisit this idea in Chapter 3. The vectors $v_1, v_2, \ldots, v_m$ in the Arnoldi decomposition form an orthonormal basis of the subspace in question. They are orthonormal by construction and a straightforward inductive argument shows that $\mathcal{K}_m(A, v) = \text{span}\{v_1, \ldots, v_m\}$.

In exact arithmetic, there are several algorithmic variants of Arnoldi's method to construct this orthonormal basis. We review some well established results on a few variants in finite precision arithmetic to motivate our current work. One variant is to

use the Gram-Schmidt process to sequentially orthogonalize each new vector against the previously orthogonalized vectors. The Arnoldi method using *Classical* Gram-Schmidt (CGS) to compute an orthonormal basis of $\mathcal{K}_m(A, v)$ given A, v, and m is given in Algorithm 2.1.1. The CGS variant is unstable. A mathematically equivalent

---

Algorithm 2.1.1: Arnoldi - CGS

Input: $A \in \mathbb{C}^{n \times n}$, $v \in \mathbb{C}^n$ and m

Result: Construction of V_{m+1} and $\widetilde{H}_{m+1}$

$v_1 = v/||v||_2$;

for $j = 1 : m$ **do**

- $h_j = V_j^H A v_j$;
- $v = Av_j - V_j h_j$;
- $h_{j+1,j} = ||v||_2$;
- **if** $h_{j+1,j} = 0$ **then**
  - stop;
- $v_{j+1} = v/h_{j+1,j}$; $V_{j+1} = [V_j, v_{j+1}]$;
- $\widetilde{H}_j = \begin{bmatrix} \widetilde{H}_{j-1} & h_j \\ 0 & h_{j+1,j} \end{bmatrix}$;

---

but numerically more attractive version of CGS is the subtle rearrangement called *Modified* Gram-Schmidt (MGS). Either approach requires $2mn^2$ flops for the matrix-vector multiplications (we assume the matrix A to be dense) and $2m^2n$ flops for the vector operations (due to the Gram-Schmidt process). Though MGS is numerically more attractive than the CGS variant, it still inherits the numerical instabilities of the Gram-Schmidt process. The orthogonality of the columns of V_m can severely be affected by roundoff errors. To remedy this, there are a few computationally attractive alternatives. The vector v_{j+1} can be reorthogonalized against the columns of V_j whenever one suspects loss of orthogonality may have occurred. This improves

stability, but the process has its limitations and this adds flops. Extensive details on the Gram-Schmidt process may be found in [45]. Another option is to approach the process of orthogonalization in an entirely different way.

The final variant under consideration changes the orthogonalization scheme and uses one of the most reliable orthogonalization procedures, one based on Householder transformations. As Trefethen and Bau explain it, "while the Gram-Schmidt process can be viewed as a sequence of elementary triangular matrices applied to generate a matrix with orthonormal columns, the Householder formulation can be viewed as a sequence of elementary unitary matrices whose product forms a triangular matrix." The Householder variant has very appealing properties, specifically the use of orthogonal transformations guarantees numerical stability. This does come with an increase in the number of flops. The use of Householder in the context of Arnoldi is backward stable but requires $4m^2n - \frac{4}{3}m^3$ flops (this does not count the $2mn^2$ flops for the matrix-vector multiplications). The Arnoldi procedure with Householder makes use of reflectors of the form $P = I - \frac{2}{x^H x} x x^H$ which introduce zeros in desired locations. Walker [67] initially formulated the method in the context of solving large nonsymmetric linear systems.

As we desire an approach rich in BLAS Level 3 operations, we turn our attention to block methods that use blocks of vectors rather than single vectors. We note that there are other benefits in using a block approach in the context of an iterative eigensolver. The iterative eigensolver converges faster and is more robust in the presence of clustered eigenvalues. This will be examined in Chapter 3. The extension of the Arnoldi method to a block algorithm is straightforward. Rather than using an initial starting vector, a block of vectors, $V \in \mathbb{C}^{n \times b}$, is used and the Arnoldi procedure constructs an orthonormal basis for the block Krylov subspace

$$\mathcal{K}_{mr}(A, V) = \text{span}\{V, AV, A^2V, \dots, A^{m-1}V\}.$$

An Arnoldi decomposition now takes the form

$$AW_m = W_m H_m + V_{m+1} H_{m+1,m} E_m^H \tag{2.3}$$

where each $V_i \in \mathbb{C}^{n\times b}$ and

$$\begin{aligned} W_m &= [V_1, V_2, \ldots, V_m], \\ E_m &= \text{matrix of the last } b \text{ columns of } I_{mr}. \end{aligned}$$

Here I_{mb} is the $mb \times m$ identity matrix, $W_m \in \mathbb{C}^{n\times mb}$ has orthonormal columns and $H_m \in \mathbb{C}^{mb\times mb}$ is band-Hessenberg as there are b subdiagonals. For simplicity we write

$$AW_m = W_{m+1} \widetilde{H}_{m+1} \tag{2.4}$$

where $\widetilde{H}_{m+1} \in \mathbb{C}^{(m+1)b\times mb}$ is the block version of our simplified notation given by

$$\widetilde{H}_{m+1} = \begin{bmatrix} H_m \\ H_{m+1,m} E_m^H \end{bmatrix}.$$

A block analog of Algorithm 2.1.1 follows immediately but some variants are possible depending on concerns for parallelism as detailed in [55]. The block version of Arnoldi with Householder fits nicely in the context of BLAS Level 3 operations thanks to the compact WY representation presented in Schreiber and Van Loan [56]. In the compact WY form, a product of b Householder reflectors can be represented as

$$Q = I_n - YTY^T \tag{2.5}$$

where $Y \in \mathbb{C}^{n\times b}$ is a lower trapezoidal matrix and $T \in \mathbb{C}^{b\times b}$ is an upper triangular matrix. This enables the use of BLAS Level 3 operations. This performance along with the aforementioned backward stability is why we opt to use Householder orthogonalization in the Arnoldi method.

2.2 Tiled Arnoldi with Householder

Block formulations of the Arnoldi method are not new. Morgan [51] developed one in in the context of solving linear systems with his introduction of GMRES-DR that uses Ruhe's variant of block Arnoldi. Explicit formulation of Ruhe's variant can be found in [55]. In the context of the NEP, Möller [49] and Baglama [7] offer different implementations for each of their approaches. We present a new implementation with the focus on performance in the context of multicore architectures that works on tiles. This also sets the foundation for our ensuing work on algorithms for the NEP.

The algorithmic formulation of Arnoldi with Householder follows directly from employing the compact WY representation. To simplify the presentation we adopt a Matlab style notation. It will be convenient to refer to locations within a matrix as follows: for an $n \times n$ matrix A, $A(7 : n, 1 : b)$ denotes the submatrix consisting of the first b columns and last $n - 6$ rows. As we are working with blocks, it will simplify the notation to refer to the jth block of rows or columns within a matrix. For example, for the jth column block of $W_m = [V_1, V_2, \ldots, V_m]$ we may write $W_m(:, \{j\})$ as $W_m(:, \{j\}) = W_m(1 : n, (j - 1)b + 1 : jb)$.

Algorithm 2.2.1 outlines the essential steps to compute an orthonormal basis for $\mathcal{K}_{mb}(A, U)$ given $A \in \mathbb{C}^{n\times n}$, starting block $U \in \mathbb{C}^{n\times b}$ and m. Algorithm 2.2.1 is formulated to employ specific existing computational kernels in LAPACK 3.4.1, as denoted in parentheses next to each major computation. We will discuss the LAPACK subroutines used as they form the basis for our new implementation. The central computational kernel in Algorithm 2.2.1 is the QR factorization. The blocked QR factorization with compact WY version of $Q = I_n - YTY^T$ is accomplished by the LAPACK xGEQRT subroutine. The call to xGEQRT constructs a compact WY representation of b Householder reflections that introduce zeros in b consecutive

Algorithm 2.2.1: Block Arnoldi - HH

Input: A, U and m

Result: Construction of Q_{m+1} and $\widetilde{H}_{m+1}$ such that $AQ_m = Q_{m+1}\widetilde{H}_{m+1}$

```
U = AU (xGEMM); Q = I_{n×mb} first mr columns of I_n;
for j = 0 : m − 1 do
    Compute the QR factorization of U(jb + 1 : n, :) (xGEQRT),
    generating T and storing the reflectors in V(:, {j + 1}));
    Accumulate reflectors to explicitly build next block of Q (xGEMQRT);
    for k = j : −1 : 1 do
        Q(:, {j + 1}) = Q(:, {j + 1}) − V(:, {k})T(:, {k})V(:, {k})^H Q(:, {j + 1});
    if j < m − 1 then
        U = AQ(:, {j + 1}) (xGEMM);
        Update U with reflectors from previous columns (xGEMQRT);
        for k = 1 : j + 1 do
            U = U − V(:, {k})T(:, {k})^H V(:, {k})^H U;
```

columns. The Y factor in this case is unit lower triangular by construction and to save on storage, the reflectors that construct the Y factor, aside from the 1's on the diagonal, are stored in the zeroed out area of the matrix passed to xGEQRT. We denote the essential pieces of the Y factor that we must store by V_{m+1}. This allows for compact storage of the $\widetilde{H}_{m+1}$ factor and the essential pieces of Y in the matrix V_{m+1} as seen in Figure 2.1. For each block in the Krylov sequence, an upper triangular block reflector is computed and the final T generated here has the form $T = [T_1, T_2, \ldots, T_m]$ where each T_j is a $b \times b$ upper triangular matrix. It is worth noting that the T factor here has a slightly different form than a full $mb \times mb$ factor in a compact WY representation as LAPACK is designed to efficiently exploit the block structure. Applying the compact WY version of the reflectors efficiently when

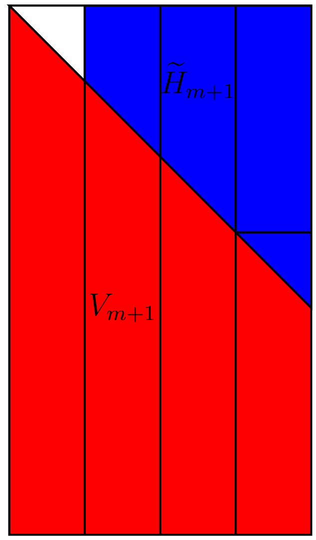


Figure 2.1: Compact storage of the Arnoldi decomposition

multiplying by Q or Q^H is handled by xGEMQRT which uses the reflectors stored in the lower part of V_{m+1} and the triangular factors in T. Multiplication of the new block of vectors by matrix A requires the BLAS xGEMM subroutine.

LAPACK subroutines are designed to use blocks of columns, or blocks of rows, to cast the operations as matrix multiplications rather than vector operations. This facilitates Level 3 BLAS operations, but there are issues that limit performance. Of note are the synchronizations performed at each step and the lack of fine grain tasks for increased parallelism. Multithreaded BLAS can be utilized, but this is often not enough to ensure that these algorithms perform optimally in the context of multicore.

To take full advantage of emerging computer architectures, we must reformulate our block algorithm. Multicore architectures require algorithms that can take full advantage of their structure. To this end, algorithms have been moving towards the class of so-called tile algorithms [19, 18, 53] and are available as part of efforts like the PLASMA library. The data layout in the context of PLASMA requires a matrix to be reordered into smaller regions of memory, called tiles. A matrix $A \in \mathbb{C}^{n\times n}$

can be subdivided into tiles ranging in size from 1×1 tiles to $n \times n$ tiles, but once set, the tile size is fixed for the duration of the algorithm. Finding the tile size that achieves optimal performance requires a bit of tuning, so for now, we will assume A is decomposed into n_t tiles of size $n_b \times n_b$ so that $n = n_t n_b$. We will add one more notational convenience for our algorithms and let $A_{i,j}$ denote the $n_b \times n_b$ tile in the ith row and jth column of the tiling of A. It is important to note that currently only square tiles are permitted in PLASMA and our application will require us to make accommodations for various tile sizes. The decomposition of a matrix into tiles and the subdivision of computational tasks adds a new dynamic to our problem in that these tasks must be organized. As detailed by Bouwmeester [13], working with tiles forces us to consider several specific features. Tiles can be in three states, namely zero, triangle or full, and introducing zeros in a matrix can be accomplished by three different tasks. Tile algorithms allow the separation of factorization stages and corresponding update stages whereas these are considered a single stage in coarse-grain algorithms, such as those in LAPACK.

Organizing the factorization tasks and the decoupled associated updates in different ways leads to different algorithmic formulations and possibly different performance. Computations are often expressed through a task graph, often called a Directed Acyclic Graph (DAG), where nodes are elementary tasks that operate on tiles and edges represent the dependencies. Different algorithmic formulations may result in different DAGs. To compare various algorithmic formulations, we look at the respective DAGs and compute the critical path. The critical path is the longest necessary path from start to finish and represents a sequence of operations that must be carried out sequentially. Analyzing DAGs and critical paths allows for the selection of optimal parallelization strategies. Much work is currently being devoted to developing scheduling strategies that improve performance. After introducing the essential computational kernels of our algorithm, we will revisit the question of performance.

Extending our algorithm from the LAPACK framework to that of PLASMA requires replacing each major kernel with a tiled analog. The block QR factorization with compact WY form performed by xGEQRT (line 3 of Algorithm 2.2.1) will be replaced with a tiled QR factorization. Tiled QR factorizations for multicore were introduced in [19, 18, 53] and recently improved and analyzed by Bouwmeester [13]. Line 3 of Algorithm 2.2.1 requires the QR factorization of a tall and skinny matrix, one that has many more rows than columns. As we will see later in Chapter 4, the number of columns b in a block Krylov subspace method will be set by practical concerns and, in general, it will be "small", much smaller than the recommended tile size n_b.

The basic structure of our approach, Algorithm 2.2.2, remains the same but the details of each step require a bit of discussion. From $A \in \mathbb{C}^{n\times n}$ and a starting block $U \in \mathbb{C}^{n\times b}$, we compute a block Arnoldi decomposition of size m. Our approach begins with the $n \times mb$ identity matrix in Q. Next we compute the product AU and as depicted in Line 3 of Algorithm 2.2.2. Here we explicitly listed the double loop to give the flavor of tile operations. Continuing with a detailed tile perspective is not feasible for a readable pseudocode, so we expand on each step with a more detailed discussion.

As before, the primary computation in Algorithm 2.2.2 is the QR factorization. In the previous case, the LAPACK routine xGEQRT zeroed out b columns at a time and computed the corresponding block Householder reflector in compact WY form. In the tiled version, the structure of the QR factorization changes with the opportunities to increase parallelism and this leads to new computational tasks among the tiles. For our application, we will consider the case where we wish to compute the QR factorization of a matrix $V \in \mathbb{C}^{n_t n_b \times b}$, that is, V is comprised of one column of n_t

Algorithm 2.2.2: Tiled block Arnoldi with HH

Input: A, U, m, b, and n_t

Result: Construction of Q_{m+1} and $\widetilde{H}_{m+1}$ such that $AQ_m = Q_{m+1}\widetilde{H}_{m+1}$

```
Q = I_{n×mb} first mb columns of I_n; Q(:, {1}) = U;
for k = 0 : m − 1 do
    for i = 0 : n_t − 1 do
        for j = 0 : n_t − 1 do
            V_{i,k} ← A_{i,j} Q_{j,k−1} (xGEMM);
    Update V with reflectors from previous factorizations (xUNMQR,
    xTTMQR);
    Compute the QR factorization of V(jb + 1 : n, {j + 1}) (xGEQRT,
    xTTQRT);
    Accumulate reflectors to explicitly build next block of Q (xTTMQR and
    xUNMQR);
```

tiles each of size $n_b \times b$ as in the following

$$V = \begin{bmatrix} V_{1,1} \\ V_{2,1} \\ \vdots \\ V_{n_t,1} \end{bmatrix}, \text{ with } V_{i,1} \in \mathbb{C}^{n_b \times b}, i = 1, \ldots, n_t.$$

Computing the QR factorization of a column of tiles allows for different algorithmic formulations based on the ordering of computations on each tile. Using existing PLASMA routines, we could first compute the QR factorization of $V_{1,1}$ so that we

have

$$V = \begin{bmatrix} T_{1,1} \\ V_{2,1} \\ \vdots \\ V_{n_t,1} \end{bmatrix},$$

where $T_{1,1}$ is upper triangular with the Householder reflectors stored below the diagonal as is standard in LAPACK. Then we could use $T_{1,1}$ to systematically zero out the remaining tiles. Each of these elimination steps has the form of a triangle on top of a square and can be accomplished by PLASMA's xTSQRT routine. Updates would then require both xUNMQR and xTSMQR. The process just outlined can be described by using a flat tree beginning at the top. This approach is sequential and does not parallelize. Fortunately, the tiled environment allows for more choices.

For example, we could proceed by first computing the QR factorizations of each of the n_t tiles, $V_{i,1}$ with $i = 1, \ldots, n_t$, using n_t calls to PLASMA's xGEQRT. This results in a column of n_t upper triangular factors with respective Householder reflectors stored below the diagonal of each of the n_t tiles. Each individual tile now has the same structure as the output of LAPACK's xGEQRT. Next, we could proceed by using the triangular factors to eliminate the triangular factors directly below. This approach gives rise to a computational kernel, PLASMA's xTTQRT detailed in [13], that zeroes a triangle with a triangle on top. Repeating this step we could proceed left to right as depicted in Figure 2.2 where the steps are organized using a binomial tree. The QR factorization depicted in Figure 2.2 requires different routines to update using the Householder reflectors. Each call to xGEQRT generates compact WY components that may be applied by calling PLASMA's xUNMQR. In the case of xTTQRT, the corresponding update is achieved by the PLASMA routine xTTMQR. We will explore some variants of Algorithm 2.2.2 shortly, but comprehensive information on various elimination lists and algorithmic formulations of tiled QR may be found in [13].

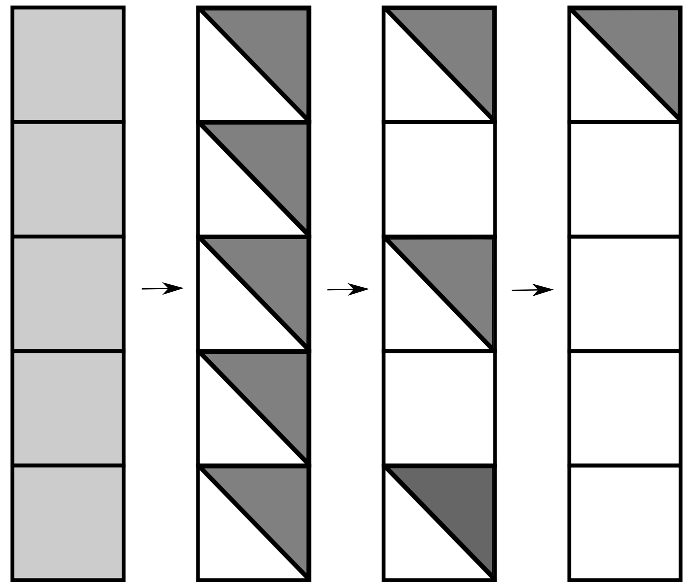


Figure 2.2: Tiled QR with $n_t = 5$ using xTTQRT

After computing the QR factorization of the column, we build the next column block in the matrix Q. To do so, we must apply the reflectors, that is apply the updates, in reverse order requiring the use of both xTTMQR and then xUNMQR. By reverse order here we mean that the updates must use the same elimination tree used in the forward sense. If we use a binomial tree going in the forward direction as in Figure 2.2 then we must traverse that same tree in the reverse direction. We then begin the loop again and multiply our newly computed column block of Q by A. This must then be updated with the reflectors from previous columns using xUNMQR and then xTTMQR. We are now ready to compute the QR factorization of the next column of tiles and continue on with the Arnoldi process.

What is not evident in Figure 2.2 is the case when the single block of columns does not fit nicely in the context of square tiles. As PLASMA requires square tiles, we had to modify several existing routines so that they could operate on sub-tiles. Routines that were modified include the QR factorization xGEQRT with associated update xUNMQR and the zeroing out of a triangle with a triangle xTTQRT with

associated update xTTMQR. An example modification is presented in Figure **??** for the routine QUARK_CORE_ZGEQRT. The last two lines, not including the 0, “lock” the tile referenced by Ap and the corresponding Tp tile since the reference to A and T point within the tile and do not “lock” the entire tile. We needed to point to within the tile but this does not indicate that a dependency is needed.

```
#include ‘‘doth.h’’
void my_QUARK_CORE_zgeqrt(Quark *quark, Quark_Task_Flags *task_flags,
                       int m, int n, int ib, int nb,
                       PLASMA_Complex64_t *A, int lda,
                       PLASMA_Complex64_t *T, int ldt,
                       PLASMA_Complex64_t *Ap, PLASMA_Complex64_t *Tp)
{
    DAG_CORE_GEQRT;
    QUARK_Insert_Task(quark, CORE_zgeqrt_quark, task_flags,
        sizeof(int),                        &m,     VALUE,
        sizeof(int),                        &n,     VALUE,
        sizeof(int),                        &ib,    VALUE,
        sizeof(PLASMA_Complex64_t)*nb*nb,    A,             INOUT,
        sizeof(int),                        &lda,   VALUE,
        sizeof(PLASMA_Complex64_t)*ib*nb,    T,             OUTPUT,
        sizeof(int),                        &ldt,   VALUE,
        sizeof(PLASMA_Complex64_t)*nb,      NULL,           SCRATCH,
        sizeof(PLASMA_Complex64_t)*ib*nb,   NULL,           SCRATCH,
        sizeof(PLASMA_Complex64_t)*nb*nb,    Ap,            INOUT,
        sizeof(PLASMA_Complex64_t)*ib*nb,    Tp,            OUTPUT,
        0);
}
```

Figure 2.3: Modification of QUARK_CORE_ZGEQRT for sub-tiles

2.3 Numerical Results

Here we compare different formulations of our tiled Arnoldi method by adjusting the underlying elimination list. We assume an unlimited number of processors in this analysis and investigate a few algorithmic variations. As the number of resources is unlimited any task may be executed as soon as the dependencies are satisfied. Algorithm 2.2.2 requires a QR factorization of a single column of tiles but also uses

the updates to explicitly form the Q factor one block column at a time and update new columns using previously computed reflectors. By using different elimination trees for the QR decomposition, the tree used for the update changes as well. This in turn changes the DAG and possibly allows for more interleaving of the various steps, i.e., the matrix multiplication and factorization, and might also provide better pipelining of update and factorization trees.

Here we present some numerical experiments comparing our tiled code with two different trees (binomial and pipeline) to a reference implementation. We present results for a diagonal matrix, a tridiagonal matrix and a dense matrix. The Rooftop

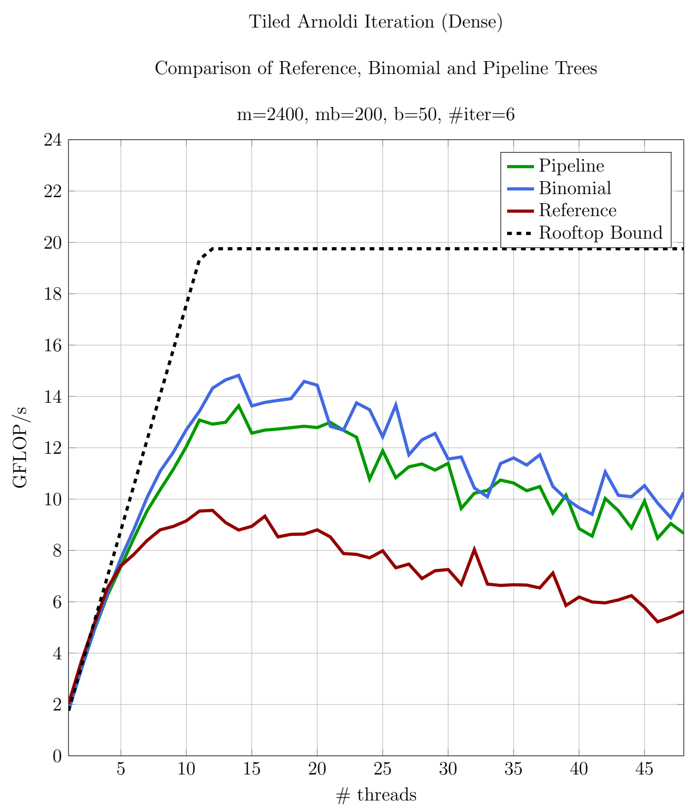


Figure 2.4: Performance comparison for a dense (smaller) matrix

Bound is an upper bound on performance on p processors. To calculate the Rooftop

Bound, we use

$$\text{Rooftop}(p) = \max\left(p, \frac{\text{total \# flops}}{\text{\# of flops on the critical path}}\right) \cdot (\text{performance of a processor}).$$

In Figure 2.4, A is a dense, $2,400 \times 2,400$ matrix, the block size is $b = 50$, the initial vector V is $2,400 \times 50$, and we want to perform 20 Arnoldi iterations so that we obtain a band Hessenberg matrix of size $1,000 \times 1,000$ with bandwidth 50. We present a reference implementation which is a monolithic implementation calling LAPACK and BLAS. For our tile implementation and this experiment, the tile size for A is $m_b = 200$, so that $n_t = 12$, this choice makes V to be 12×1 in term of tiles with 200×50 tiles. As explained, the matrix V is made of rectangular tiles and PLASMA does not natively handle rectangular tiles so we needed to hack rectangular matrix support into the PLASMA library. To obtain square tiles, we could have tiled A with 50×50 tiles however we judged that there were too many drawbacks in tiling A with a tile size equal to the size of the block in the Krylov method. (i) We expect the block size of block Arnoldi to vary given an iterative method. We do not want to change the data layout of A over and over. (ii) The block size of block Arnoldi is determined for behaving well for the eigensolver, while the tile size is determined for performance. In general, we expect the block size of block Arnoldi to be smaller than the tile size. Given this setup, $m = 2,400$ and $b = 50$ and #iter=20, the total work for the Arnoldi factorization is approximately 4.35 Gflops and the length of the weighted critical path for the binomial tree method is 0.39 Gflops. We use the `ig` multicore machine located in Tennessee for our experiments. The performance of one core is 1.757 Gflop/sec. In Figure 2.4, we plot the Rooftop Bound for the binomial tree. Note that the point at which the performance starts to level off is the same point at which the Rooftop Bound reaches it maximum (in terms of number of processors). We acknowledge that there is still a large gap between the bound and the curves. We would like to have more descriptive upper bounds.

We could not produce a Rooftop Bound for the case where $k = 20$ and $m = 9,600$ in Figure 2.5 since we have no closed-form formula for the weighted critical path length and the DAG was too large for the computer to calculate it. The results for

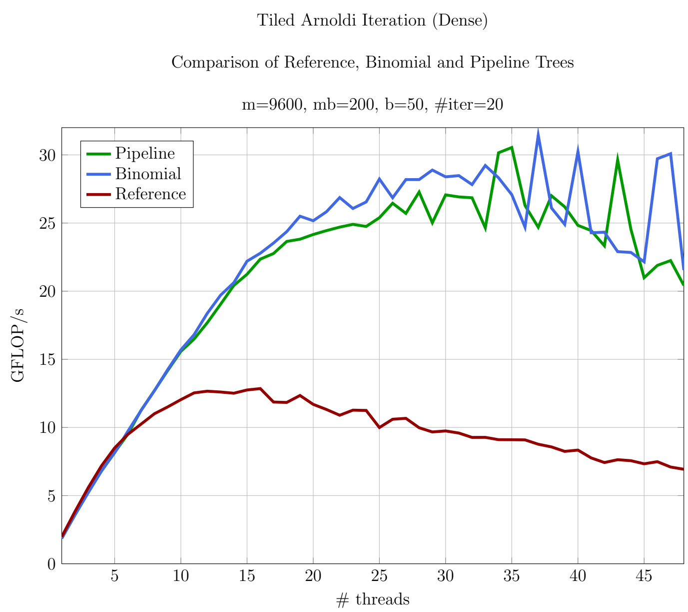


Figure 2.5: Performance comparison for a tiled dense (larger) matrix

the tridiagonal case can be seen in Figure 2.6 and for the diagonal case in Figure 2.7. In all of four cases we conclude that our Arnoldi implementation performs better than the reference implementations.

2.4 Conclusion and Future Work

We have proposed a new algorithm to compute a block Arnoldi expansion with Householder reflectors on multicore architectures. An experimental study has shown that our algorithm performs significantly better than our reference implementations. We would like to obtain closed-form formula for the critical path of our new algorithm, and we would like to benchmark our code with sparse matrices.

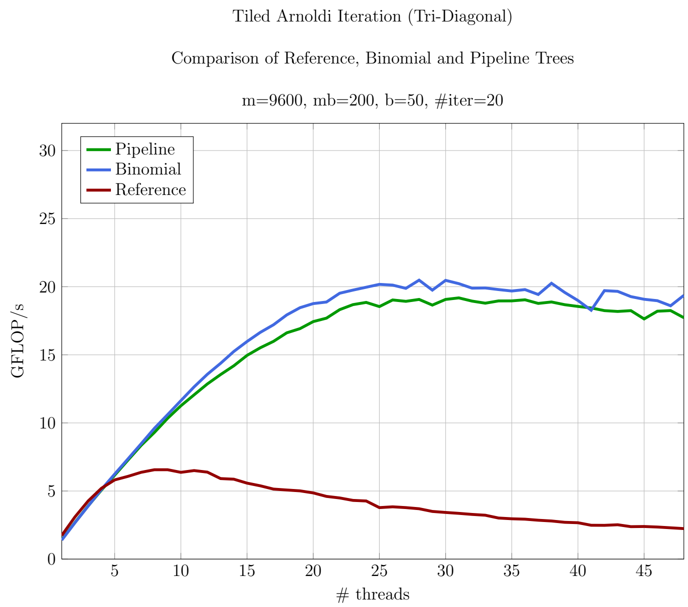


Figure 2.6: Performance comparison for a tiled tridiagonal matrix

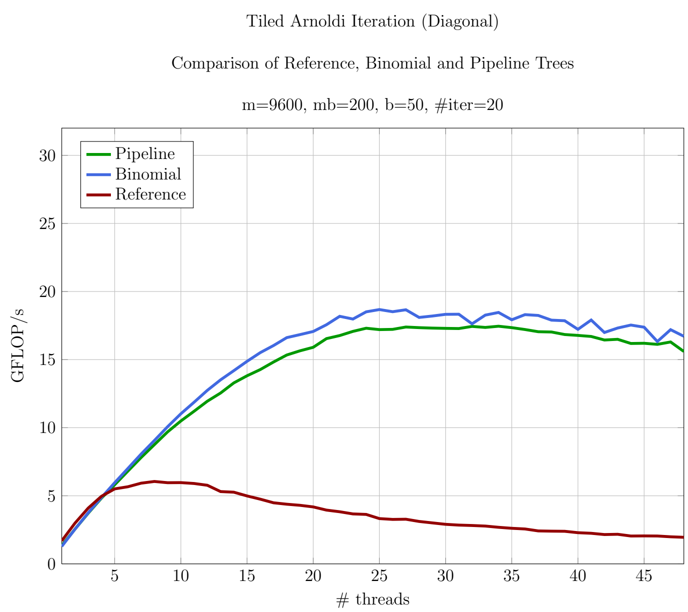


Figure 2.7: Performance comparison for a tiled diagonal matrix

3. Alternatives to the QR Algorithm for NEP

In Chapter 2, we studied an efficient Arnoldi tile expansion for multicore system. In this chapter, we turn our focus to the computation of eigenvalues and associated invariant subspaces. This chapter motivates our work in Chapter 4 where we focus solely on the block Krylov-Schur method. We are specifically interested in computing a partial Schur decomposition as in Equation 1.3 using an "iterative method" algorithm. Here we detail our block extensions of various approaches and undertake an experimental numerical study for various algorithms in the context of computing any number of eigenvalues of a nonsymmetric matrix. Our implementations of several approaches are compared to existing unblocked implementations and blocked codes when available. All algorithms are implemented in MATLAB. When applicable, we survey current state-of-the-art implementations and related software libraries.

Here we focus strictly on methods aiming to compute eigenvalues of an $n \times n$ matrix A which work by accessing the operator A only through "matrix-vector products". In particular, none of the algorithms considered in this Chapter reduces the matrix to Hessenberg form. There are several reasons for this design choice. Though the reduction to Hessenberg form is the first phase of practical implementations of the QR algorithm, it is a costly endeavor, in particular in terms of communication and parallelism. Using Householder reflectors, proceeding a column at a time, this computation requires approximately $\frac{10}{3}n^3$ flops and is based mainly on Level 2 BLAS operations.

The accumulation of Householder reflectors into compact WY form [56] can be used to improve the situation by incorporating Level 3 BLAS operations when possible. This was described by Dongarra, Hammarling, and Sorensen [23], but performance issues still remain in the context of multicore. Recently, Quintana-Ortí and van de Geijn [54] cast more of the required computations in efficient matrix-matrix operations achieving significant performance improvements. Yet, 20% of the flops

remain in Level 2 BLAS operations. Howell and Diaa [36] presented an algorithm, BHESS, using Gaussian similarity transformations to reduce a general real square matrix to a small band Hessenberg form. Eigenvalues can then be computed using the bulge-chasing BR iteration or the Rayleigh quotient iteration. The overall cost of the BHESS-BR method was reported to be typically $\frac{8}{3}n^3$ flops compared to the standard QR iteration which requires $\frac{10}{3}n^3$ for the reduction and 10 to $16n^2$ flops for the ensuing iteration on the Hessenberg factor. The BHESS-BR approach is appropriate for computing nonextremal eigenvalues of mildly nonnormal matrices [36].

A two-staged approach, described by Ltaief, Kurzak, and Dongarra [48], showed that an initial reduction to block Hessenberg form, also called band Hessenberg as there is a block or band of subdiagonals rather than one subdiagonal, is efficiently handled by a tile algorithm variant. Using the tiled approach, their algorithm with Householder refectors achieves 72% (95 Gfop/s) of the DGEMM peak on a $12,000 \times 12,000$ matrix size with 16 Intel Tigerton 2.4GHz cores and most of the operations are in Level 3 BLAS. The second phase of the proposed method [48] reduces the block Hessenberg matrix to Hessenberg form using a bulge chasing approach similar to some extent to what is done in the QR algorithm. The algorithm used in the second phase does not achieve any comparable performance, mainly due to the inefficiency of the parallel bulge chasing procedure on multicore architectures. The bulge chasing in the second phase may benefit from the optimal packing strategy [39], but we do not investigate this further. Because of these challenges, we turn our focus to iterative methods (Arnoldi, Jacobi-Davidson) which avoid complete reduction to Hessenberg form. As we will see though, the implicit QR algorithm will remain an essential piece of any approach to the NEP.

We note that, if we were to run the Block Arnoldi process presented in Chapter 2 to completion (n steps), we would perform a Block Hessenberg reduction as in [39]. However, the block Hessenberg matrix we obtain is associated with a Krylov space

expansion, and so early submatrices of this matrix should contain relevant information on invariant subspaces of the initial matrix.

3.1 Iterative Methods

3.1.1 Arnoldi for the nonsymmetric eigenvalue problem and IRAM

The Arnoldi procedure discussed in Chapter 2 not only produces an orthonormal basis for the subspace $\kappa_m(A, v)$, but it also generates information about the eigenvalues of A. Though originally developed as a method to reduce a general matrix to upper Hessenberg form, the Arnoldi method may be viewed as the computation of projections onto successive Krylov subspaces. In exact arithmetic, Algorithm 2.1.1 will terminate on line 6 if $h_{j+1,j} = 0$, as the columns of V_j span an invariant subspace of A. In this case, the eigenvalues of H_j are the exact eigenvalues of the matrix A. If it does not terminate early, the algorithm constructs an Arnoldi decomposition of order m given by

$$AV_m = V_m H_m + \beta_m v_{m+1} \mathbf{e}_m^T. \tag{3.1}$$

Except for a rank one perturbation, we have an approximate invariant subspace relationship, that is $AV_m \approx V_m H_m$. From Equation 3.1 and the orthogonality of the columns of V_{m+1}, we have that

$$V_m^H A V_m = H_m,$$

and the approximate eigenvalues provided by projection onto the Krylov subspace $\kappa_m(A, v)$ are simply the eigenvalues of H_m. These are often called *Ritz values*, as this projection can be scene as a part of the Rayleigh-Ritz process. *Ritz vectors*, or approximate eigenvectors of A, are simply are the associated eigenvectors of H_m premultiplied by V_m. To find the eigenvalues and eigenvectors of H_m, which is already in upper Hessenberg form, we simply apply a practical version of the implicitly shifted QR algorithm.

In practice, a suitable order m for desired convergence is not known a priori and it may not be desirable to store the $n \times (m+1)$ matrix V_{m+1} as m must usually be rather large for an acceptable approximation to be computed. To address this, several restarting strategies have been suggested for how to select a new starting vector v_1. Explicit restarting strategies compute an approximate eigenvector associated with an eigenvalue of interest, say the one with largest real part. If the approximation is satisfactory, the iteration stops, and if not, the approximate eigenvector is then used as the starting vector for a new mth order Arnoldi factorization. A similar strategy may be used when multiple eigenpairs are desired. One may restart with a linear combination of approximate eigenvectors, or one may use a deflation strategy. Morgan's analysis [50] suggested that restarting with a linear combination is ill-advised unless care is taken to avoid losing accuracy when forming the linear combination. An approach to combine Ritz vectors that prevents loss of accuracy is that of Sorensen, which we will discuss in detail momentarily. As there is no easy way to determine an appropriate linear combination, we opt for a strategy based on deflating eigenpairs. We will expand on this idea when we formulate our block variant of Arnoldi for the NEP.

One of the more successful approaches based on Krylov subspaces is that of Sorensen, the implicitly restarted Arnoldi method (IRAM) presented in [59]. This method uses the implicitly shifted QR algorithm to restart the Arnoldi process. From the decomposition 3.1, for fixed k, $m-k$ shifts, $\mu_1, \ldots, \mu_{m-k}$, are selected and used to perform $m-k$ steps of the implicitly shifted QR algorithm on the Rayleigh quotient H_m. This results in

$$AV_m^+ = V_m^+ H_m^+ + \beta_m v_{m+1} \mathbf{e}_m^T Q \tag{3.2}$$

where $V_m^+ = V_m Q$, $H_m^+ = Q^H H_m Q$, and $Q = Q_1 Q_2 \cdots Q_k$ where each Q_i is the orthogonal matrix associated with each of the k shifts. Sorensen observed that the first $k-1$ components of $e_m^T Q$ are zero and that equating the first k columns on each

side yields an updated kth order Arnoldi factorization. The updated decomposition given by

$$AV_k^+ = V_k^+ H_k^+ + \beta_k v_{k+1}^+ \mathbf{e}_k^T \tag{3.3}$$

with updated residual $\beta_k v_{k+1}^+$ is a restart of the Arnoldi process with a starting vector proportional to $p(A)v_1$ where $p(A) = (A - \mu_1 I) \cdots (A - \mu_{m-k} I)$. Using this as a starting point, Sorensen continued the Arnoldi process to return to the original mth order decomposition. Sorensen showed this process may be viewed as a truncation of the implicitly shifted QR iteration. Along with this formulation, Sorensen also suggested shift choices that help locate desired parts of the spectrum. Sorensen's approach is the foundation for the ARPACK library of routines that implement IRAM [47]. In Matlab, the function **eigs** provides the user interface to ARPACK. A parallel version, PARPACK, is available as well but both offerings only compute a partial Schur decomposition of a general matrix A. As discussed earlier, one of our computational goals is the ability to compute partial and full Schur decompositions using the same approach.

3.1.2 Block Arnoldi

As always, we desire to cast our computation in Level 3 BLAS operations as much as possible for efficiency concerns, but there are other reasons. Block methods are better suited for handling the computation of clustered or multiple eigenvalues, and block methods are appropriate when more than one good initial vector is known. We will examine this more closely in our numerical experiments. Various block approaches to eigenvalue problems may be found in Saad's book [55] or in the "template" book [71]. In the case of IRAM, a block extension, bIRAM, was presented by Lehoucq and Maschhoff [46]. The bIRAM implementation compared favorably to other block variants of Arnoldi studied by Scott [57]. Of specific interest to our endeavors, the implicitly restarted block scheme was superior to block approaches

using explicit restarting strategies and also outperformed approaches using preconditioning. This includes strategies such as Chebyshev acceleration and Chebyshev preconditioning. Also of note, all implementations studied in [57] computed only a partial Schur decomposition. Currently, ARPACK does not include the bIRAM approach. One reason for this may be the complexities of such an implementation. An example of one of the difficulties in implementing such an approach is the shift strategy as discussed by Baglama [7]. The generalization to a block method creates possible convergence issues if the shifts are not chosen appropriately. Additionally, IRAM, as Stewart [64] points out, and subsequently bIRAM requires the structure of the Arnoldi decomposition to be preserved which may make it difficult to deflate converged Ritz vectors. Due to these issues, we opt to examine the behavior of a basic block Arnoldi approach. This will serve as starting point for our analysis of block methods.

Building off our work in Chapter 2, we formulate a block Arnoldi approach using Householder reflectors. Our approach uses explicit restarts and deflation to lock converged Schur vectors and is modeled after algorithm 7.43 in [71] which is reproduced in Algorithm 3.1.1. The converged eigenvalues and Schur vectors are not touched in subsequent steps of Algorithm 3.1.1. This is referred to as "hard locking" as opposed to "soft locking" in which the converged Schur vectors are continuously updated regardless of residual values. This was introduced by Knyazev [41] in the context of iterative methods for the symmetric eigenvalue problem. The upper triangular portion of the matrix H is also locked. In subsequent steps, the approximate eigenvalues are the eigenvalues of the $m \times m$ matrix H whose $k \times k$ principal submatrix is upper triangular. By locking converged eigenvalues and computed Schur vectors we are implicitly projecting out the invariant subspace already computed.

In the following, we outline the main components of one sweep of our block Arnoldi method, Algorithm 3.1.2, and discuss the specifics of our implementation.

Algorithm 3.1.1: Explicitly Restarted Arnoldi Method with Deflation

1 Set $k = 1$;
2 **for** $j = k : m$ **do**
3     $w = Av_j$;
4   Compute a set of j coefficients h_{ij} so that $w = w - \sum_{i=1}^{j} h_{ij} v_i$ is orthogonal to all previous v_i, $i = 1, 2, \ldots, j$;
5     $h_{j+1,j} = \|w\|_2$;
6     $v_{j+1} = \frac{w}{h_{j+1,j}}$;
7 Compute approximate eigenvector of A associated with the eigenvalue $\widetilde{\lambda}_k$ and its associated residual norm estimate ρ_k;
8 Orthonormalize this eigenvector against all previous v_is to get the approximate Schur vector $\widetilde{u}_k$ and define $v_k = \widetilde{u}_k$;
9 **if** *ρ_k is small enough* **then**
10   Accept the eigenvalue estimate:
11     $h_{i,k} = v_i^H A v_k, i = 1, \ldots, k$, set $k = k + 1$;
12   If the desired number of eigenvalues has been reached, stop,
13   otherwise go to **2**;
14 **else**
15   Go to **2**;

Here b denotes the block size, k_f denotes the size of the search subspace where k_f is a multiple of b, k_{max} denotes the number of desired eigenvalues, and k_{con} is the number of eigenvalues that have converged. For the sake of notation, the ensuing discussion will assume no eigenvalues have converged, that is $k_{con} = 0$. The case where $k_{con} > 0$ is detailed in the following pseudocodes. A step of the iteration begins with a block of

Algorithm 3.1.2: Block Arnoldi Method with explicit restart and deflation

Input: $A \in \mathbb{C}^{n\times n}$, $U \in \mathbb{C}^{n\times b}$, dimension of search subspace k_f and number of desired eigenvalues k_{max}

Result: Partial Schur form given by $Q_{k_{max}} \in \mathbb{C}^{n\times k_{max}}$ and $H_{k_{max}} \in \mathbb{C}^{k_{max}\times k_{max}}$

Set number of blocks: $b = \frac{k_f}{b}$;

Set $k_{con} = 0$;

while $k_{con} < k_{max}$ **do**

    Apply block Arnoldi iteration, Algorithm 3.1.3, to get a size k_f block Arnoldi decomposition:

    $AQ(:, 1 : k_{con} + k_f) = Q(:, 1 : k_{con} + k_f + b)\widetilde{H}(1 : k_{con} + k_f + b, 1 : k_{con} + k_f)$;

    Compute the Schur form of the Rayleigh quotient:

    $[Z,\ T_s] = \text{schur}\left(H(k_{con} + 1 : k_{con} + k_f, k_{con} + 1 : k_{con} + k_f),\ \text{'complex'}\ \right)$;

    Use Algorithm 3.1.6 to compute U_r that reorders the Schur form:

    $T_s \leftarrow U_r^H T_s U_r$ and $Z \leftarrow ZU_r$;

    Update the corresponding columns of Q:

    $Q(:, k_{con} + 1 : k_{con} + k_f) \leftarrow Q(:, k_{con} + 1 : k_{con} + k_f)Z$;

    Check convergence;

    if *converged* **then**

        $k_{con} = k_{con} + 1$;

        Explicitly deflate $H(k_{con} + 1 : k_{con} + b, k_{con})$;

        Collapse $Q \leftarrow Q(:, 1 : k_{con} + b)$ and build new compact WY representation using Algorithm 3.1.7;

    else

        Collapse $Q \leftarrow Q(:, 1 : k_{con} + b)$ and build new compact WY representation using Algorithm 3.1.7;

vectors $U \in \mathbb{C}^{n\times b}$ that is used to generate a size $k = \frac{k_f}{b}$ block Arnoldi decomposition

$$AQ_k = Q_{k+1}\widetilde{H}_{k+1} \tag{3.4}$$

where $\widetilde{H}_{k+1} \in \mathbb{C}^{(k_f+b)\times(k_f)}$ is given by

$$\widetilde{H}_{k+1} = \begin{bmatrix} H_k \\ H_{(k+1,k)}E_k^H \end{bmatrix}, \tag{3.5}$$

$Q_{k+1} \in \mathbb{C}^{n\times(k_f+b)}$ has orthonormal columns and a compact WY representation as in Equation 2.5, $H_k \in \mathbb{C}^{k_f\times k_f}$ is band Hessenberg, and $H_{(k+1,k)} \in \mathbb{C}^{b\times b}$. Here Q_{k+1} refers to a matrix with $k+1$ blocks of size $n \times b$ and $Q_{(k+1)}$ refers to the $n \times b$ matrix making up the $(k+1)$st block of Q_{k+1}. For the pseudocode, it will be convenient to use the Matlab style notation introduced in Chapter 2, that is $Q_k = Q(:, 1 : kb) = Q(:, 1 : k_f)$ and $Q_{(k+1)} = Q(:, kb+1 : kb+b) = Q(:, k_f+1 : k_f+b)$ or using the block notation $Q_{(k+1)} = Q(:, \{k+1\})$.

Expansion using block Arnoldi is detailed in Algorithm 3.1.3. In Algorithm 3.1.3, we use Matlab's function **qr** to compute some of the components of the compact WY representation. Using the "economy-size" option, we compute the upper triangular factor with components of the reflectors stored below the diagonal as in LAPACK. The Householder reflectors have the form $H(i) = I - \tau_i v_i v_i^H$ where v_i is unit lower triangular and thus, its upper part does not need to be stored. Since Matlab's interface does not provide the scalars, τ_i, needed to construct the elementary reflectors, we opted to implement our own xLARFG in Matlab to generate the missing components to be able to compare to LAPACK when constructing the compact WY form. Details on the computation of the scalars may be found in Algorithm 3.1.4. The scalars we compute with our xLARFG are then used in our own Matlab implementation of xLARFT to construct the triangular factor in the compact WY representation. Details may be found in Algorithm 3.1.5. We will revisit this in Chapter 4 as our algorithmic construction motivates a slight modification of xLARFT in LAPACK to

Algorithm 3.1.3: Block Arnoldi Iteration

Input: A, k, and possibly collapsed Q in compact WY form

Result: kth order Arnoldi decomposition with Q in compact WY form

if $k_{con} = 0$ **then**
    for $j = 0 : b_1$ **do**
        $V =$ qr$(U(k_{con} + jb + 1 : n, 1 : b), 0)$ and compute scalars $\tau(k_{con} + \{j+1\})$ for the elementary reflectors;
        if $j > 0$ **then**
            $H(1 : jb, \{j\}) = U(1 : jb, :)$;
            $H(\{j+1\}, \{j\}) =$ triu$(V(1 : b, 1 : b))$;
        $Y(jb + 1 : n, \{j+1\}) =$ tril$(V, -1) +$ eye(size(V)); $T =$ zlarft(Y, τ);
        $Q(:, \{j+1\}) = Q(:, \{j+1\}) - YTY^H Q(:, \{j+1\})$;
        if $j < b_1$ **then**
            $U = AQ(:, \{j+1\}))$, $U = U - YT^H Y^H U$;
else
    for $j = 1 : b_1$ **do**
        $U = AQ(:, k_{con} + \{j\})$;
        $U = U - YT^H Y^H U$;
        $U(1 : k_{con} + b, :) = R(1 : k_{con} + b, 1 : k_{con} + b)^T U(1 : k_{con} + b, :)$;
        $V =$ qr$(U(k_{con} + jb + 1 : n, :), 0)$ and compute scalars $\tau(k_{con} + \{j+1\})$ for the elementary reflectors;
        $Y(k_{con} + jb + 1 : n, k_{con} + \{j+1\}) =$ tril$(V, -1) +$ eye(size(V));
        $T =$ zlarft(Y, τ);
        $H(1 : jb + k_{con}, k_{con} + \{j\}) = U(1 : jb + k_{con}, :)$;
        $H(k_{con} + \{j+1\}), k_{con} + \{j\}) =$ triu$(V(1 : b, 1 : b))$;
        $Q(:, k_{con} + \{j+1\}) = Q(:, k_{con} + \{j+1\}) - YTY^H Q(:, k_{con} + \{j+1\})$;

Algorithm 3.1.4: MATLAB implementation of ZLARFG

Input: A vector $x \in \mathbb{C}^n$

Result: Scalars β, τ and vector v

```
n = length(x);
alpha = x(1); xnorm = norm(x(2 : n), 2);
ar = real(alpha); ai = imag(alpha);
if xnorm = 0 then
    tau = 0; beta = alpha; v = x(2:n,:);
else
    beta = -sign(ar)*sqrt(ar^2 + ai^2 + xnorm^2);
    tau = (beta - ar)/beta - ai/beta*i; v = x(2:n)/(alpha - beta);
```

Algorithm 3.1.5: Matlab implementation of ZLARFT

Input: reflectors Y and associated scalars τ

Result: triangular factor T for compact WY form

```
[n, k] = size(V);
T = zeros(k);
for i = 1 : k do
    T(1 : i-1, i) = -τ(i)T(1 : i-1, 1 : i-1)(V(:, 1 : i-1)^H V(:, i));
    T(i, i) = τ(i);
```

avoid unneeded computations. If no eigenvalues have converged, the matrix Q_{k+1} has the following form

$$Q_{k+1} = I_{n,(k+1)b} - YTY^H I_{n,(k+1)b}, \tag{3.6}$$

where $I_{n,(k+1)b}$ is the first $(k+1)b$ columns of the $n \times n$ identity matrix, $Y \in \mathbb{C}^{n\times(k+1)b}$ with unit diagonal and $T \in \mathbb{C}^{(k+1)b\times(k+1)b}$ as in LAPACK. If k_{con} eigenvalues have been deflated, Q_{k+1} has the form

$$Q_{k+1} = \widetilde{I}_{n,k_{con}+(k+1)b} - YTY^H \widetilde{I}_{n,k_{con}+(k+1)b}, \tag{3.7}$$

where $\widetilde{I}_{n,k_{con}+(k+1)b} \in \mathbb{C}^{n\times k_{con}+(k+1)b}$ has the form

$$\widetilde{I}_{n,k_{con}+(k+1)b} = \begin{bmatrix} R_{k_{con}+b} & 0_{k_{con}+b,k-k_{con}-b} \\ 0_{n-k_{con}-b,k_{con}+b} & I_{k-k_{con}-b} \end{bmatrix}.$$

Here $R_{k_{con}+b} \in \mathbb{C}^{(k_{con}+b)\times(k_{con}+b)}$ is a diagonal matrix with entries ± 1 generated when we rebuild the compact WY form after deflating or restarting and the other components are appropriately sized matrices of zeros and the identity matrix.

In the next step of our block Arnoldi approach, we compute the Schur factorization of the Rayleigh quotient matrix H_k, that is

$$H_k V_k = V_k S_k, \tag{3.8}$$

where $V_k \in \mathbb{C}^{kb\times kb}$, $V_k^H V_k = I$, and $S_k \in \mathbb{C}^{kb\times kb}$ is upper triangular. In our current implementation, S_k is upper triangular rather than upper block triangular, but we could adjust our approach to work in real arithmetic. The choice to work in complex arithmetic was made, in part, to compare to some of the available implementations of unblocked methods that use the same approach. Our implementation uses MATLAB's function **schur** which provides the interface to LAPACK's routines xGEHRD, xUNGHR, and xHSEQR.

The Schur form 3.8 is then reordered so that the desired eigenvalues appear in the top left of the matrix S_k. Reordering the Schur form will play an important

role in all of our approaches in this section. This can be accomplished in LAPACK by the routine xTRSEN for both upper triangular and block upper triangular Schur factorizations. As we are currently working in MATLAB and since our applications keep the order of the Schur factor S_k relatively manageable, we reorder the Schur form using Givens rotations and a target to locate a specific part of the spectrum, such as the eigenvalues with largest real components. Our approach was adapted from the computations presented in [24]. Depending on the order of the Schur factor S_k, we may need to adjust how we handle reordering the Schur form. Kressner discussed block algorithms for the task of reordering Schur forms in [42] and specifically addressed the applications of interest here, namely reordering Schur forms in the Krylov-Schur and Jacobi-Davidson algorithms. To fully take advantage of level 3 BLAS operations, we should adopt a similar approach. Details of our implementation can be found in Algorithm 3.1.6 and we will postpone as future work any further analysis on efficiently reordering the Schur form.

Once the Schur form is reordered, we check to see if the Ritz values are acceptable approximations of the eigenvalues of the matrix A. After reordering, our block Arnoldi decomposition has the form

$$AQ_kV_k = \left[Q_kV_k, Q_{(k+1)}\right] \begin{bmatrix} S_k \\ H_{(k+1,k)}E_k^TV_k \end{bmatrix}. \tag{3.9}$$

so that

$$AQ_kV_k - Q_kV_kS_k = Q_{(k+1)}H_{(k+1,k)}E_k^TV_k, \tag{3.10}$$

From here we can see that the quality of the Ritz value is given by

$$\|AQ_kV_k - Q_kV_kS_k\|_2 = \|Q_{(k+1)}H_{(k+1,k)}E_k^TV_k\|_2$$

Algorithm 3.1.6: Reorder Schur Form

Input: Unitary $Z \in \mathbb{C}^{k \times k}$, upper triangular $T_s \in \mathbb{C}^{k \times k}$ and a target

Result: Reordered T_s and $U_r \in \mathbb{C}^{k \times k}$

$U_r =$ eye(k);
for $i = 1 : k-1$ **do**
    $d =$ diag$(T_s(i : k, i : k))$;
    $[a, jj] = \min(\text{abs}(d - target))$;
    $jj = jj + i - 1$;
    for $t = jj - 1 : -1 : i$ **do**
        $x = [T_s(t, t+1), T_s(t, t) - T_s(t+1, t+1)]$;
        $G([2, 1], [2, 1]) = \text{planerot}(x^H)^H$;
        $T_s(1 : k, [t, t+1]) = T_s(1 : k, [t, t+1])G$;
        $T_s([t, t+1], :) = G^H T_s([t, t+1], :)$;
        $U_r(:, [t, t+1]) = U_r(:, [t, t+1])G$;

and a natural stopping criteria is when $\|H_{(k+1,k)} E_k^T V_k\|_2$ is sufficiently small. In ARPACK, given an Arnoldi decomposition of size k, that is

$$AU_k = [U_k, u_{k+1}] \begin{bmatrix} H_k \\ h_{k+1,k}\mathbf{e}^T \end{bmatrix}, \tag{3.11}$$

a Ritz value λ is regarded as converged when the associated Ritz vector $U_k w$, with $\|w\|_2 = 1$, satisfies

$$\|A(U_m w) - \lambda(U_m w)\|_2 = \left|h_{k+1,k} e_k^T w\right| \leq \max\{\mathbf{u}\|H_k\|_F, tol \cdot |\lambda|\}$$

where $\mathbf{u}$ is machine epsilon and tol is a user specified tolerance. Further details may be found in ARPACK's user guide [47]. This criteria guarantees a small backward error for λ as the estimate is an eigenvalue of the slightly perturbed matrix $(A + E)$ with

$$E = (-h_{m+1,m} e_m^T w) u_{m+1} (U_m w)^T$$

as detailed by Kressner [43]. We can extend this idea to our approach and accept the Ritz value λ, in the upper left of S_k after reordering, when

$$\|H_{(k+1,k)} E_k^T V_k(:,1)\|_2 \leq \max\left\{u\|H_k\|_F, tol \cdot |\lambda|\right\}. \tag{3.12}$$

The final step in a sweep of our block Arnoldi method is to check convergence using Equation 3.12. If an eigenvalue has converged, we explicitly deflate the converged eigenvalue in H_k and collapse Q_k to include the converged Schur vectors plus a block of size $n \times b$ to be used to restart the Arnoldi process. A compact WY representation of the truncated Q_k is computed and we begin the sweep again. Here we generate the matrix $R_{k_{con}+b}$ introduced in 3.7. If the eigenvalue approximation is not yet satisfactory, we collapse Q_k and build a compact WY representation of the previously converged Schur vectors and the additional $n \times b$ block to be used for restarting. Details of this final step can be found in Algorithm 3.1.7.

Algorithm 3.1.7: Explicit restart with possible deflation

Input: $Q(:, 1 : k_{con} + k + b)$ with compact WY representation

Result: $Q(:, 1 : k_{con} + b)$ with compact WY representation

$p = \text{norm}(H(k_{con} + \{b + 1\}), k_{con} + \{b\})Z(\{b\}, 1));$

if $p < \textit{check}$ **then**

- $k_{con} = k_{con} + 1;$
- Explicitly deflate:
  $H(k_{con}, k_{con}) = T_s(1, 1); H(k_{con} + 1 : k_{con} + b, k_{con}) = \text{zeros}(b, 1);$
- $Q = Q(:, 1 : k_{con} + b); H = H(1 : k_{con} + b, 1 : k_{con});$
- $V = \text{qr}(Q, 0)$; and compute scalars $t(1 : k_{con} + b)$ for elementary reflectors;
- $R = \text{diag}(\text{diag}(V(1 : k_{con} + b, 1 : k_{con} + b)));;$
- $Y = \text{tril}(V, -1) + \text{eye}(\text{size}(V))$; $T = \text{zlarft}(Y, t);;$
- $Q(k_{con} + b + 1 : n, k_{con} + b + 1 : k_{con} + b + k) = \text{eye}(n - (k_{con} + b), k);$

else

- **if** $k_{con} = 0$ **then**
  - $U = AQ(:, \{1\}); Q = \text{eye}(n, k + b);$
- **else**
  - $Q = Q(:, 1 : k_{con} + b); H = H(1 : k_{con} + b, 1 : k_{con});$
  - $V = \text{qr}(Q, 0)$; and compute scalars $t(1 : k_{con} + b)$ for elementary reflectors;
  - $R = \text{diag}(\text{diag}(V(1 : k_{con} + b, 1 : k_{con} + b)));;$
  - $Y = \text{tril}(V, -1) + \text{eye}(\text{size}(V))$; $T = \text{zlarft}(Y, t);;$
  - $Q(k_{con} + b + 1 : n, k_{con} + b + 1 : k_{con} + b + k) = \text{eye}(n - (k_{con} + b), k);$

3.1.3 Block Krylov-Schur

A more numerically reliable procedure than IRAM is Stewart's Krylov-Schur algorithm [64]. This reliability, along with the ease with which converged Ritz pairs are deflated and unwanted Ritz values are purged, makes the Krylov-Schur method an attractive alternative to IRAM. A step of Stewart's Krylov-Schur method begins and ends with a Krylov-Schur decomposition of the form

$$AV_k = V_k S_k + v_{k+1} b_{k+1}^H \tag{3.13}$$

where S_k is a $k \times k$ upper triangular matrix and the columns of V_{k+1} are orthonormal. For convenience, we write

$$AV_k = V_{k+1} \widetilde{S}_k, \tag{3.14}$$

where

$$\widetilde{S}_k = \begin{bmatrix} S_k \\ b_{k+1}^H \end{bmatrix}.$$

The Krylov-Schur method uses subspace expansion and contraction phases much like any approach based on Arnoldi decompositions. The expansion phase of Krylov-Schur proceeds exactly as in the typical Arnoldi approach and requires approximately the same amount of work. The real power of the Krylov-Schur approach is in the contraction phase. As we will see in detail in Chapter 4, the key aspect of the Krylov-Schur method is that a decomposition of the form 3.13 can be truncated at any point in the process providing an easy way to purge unwanted Ritz values. As the Krylov-Schur method works explicitly with the eigenvalues of S_k, the Rayleigh quotient, this approach is an exact-shift method. This is in contrast to methods that use other shifts such as linear combinations of the eigenvalues of S_k. We will revisit the details of the Krylov-Schur method in Chapter 4.

A block version of this approach has been implemented for symmetric matrices by Saad and Zhou in [72]. They included detailed pseudocodes, including how to handle rank deficient cases and adaptive block sizes. The numerical results provided by Saad and Zhou show that their implementation performs well against MATLAB's **eigs** function based on ARPACK, **irbleigs** by Baglama et al., an implementation of an implicitly restarted block-Lanczos algorithm [6], and **lobpcg**, without preconditioning, presented by Knyazev [40]. Of particular interest to our pursuits is that in this study, the Krylov-Schur based approach consistently outperformed the competitors as the number of desired eigenvalues was increased.

Though the Krylov-Schur method was presented as an attractive alternative to IRAM, there are few implementations in either unblocked or blocked form for non-symmetric matrices. Baglama's Augmented Block Householder Arnoldi (ABHA) method [7] is an implicit version of a block Krylov-Schur approach to the NEP as it employs Schur vectors for restarts. We will compare our approach to the publicly available program **ahbeigs** based on this ABHA method later. An implementation of the block Krylov-Schur method is available in the ANASAZI eigensolver package, which is part of TRILINOS [9].

As we did for our block Arnoldi approach, we now present a sweep of our block Krylov-Schur method, Algorithm 3.1.8, along with detailed computational kernels. We again formulate our approach based on Householder reflectors in compact WY form. A step of the iteration begins with $A \in \mathbb{C}^{n\times n}$, a block of vectors $U \in \mathbb{C}^{n\times b}$. For the remainder of this section, b denotes the block size, k_s denotes the starting basis size and the size of the basis after contraction, k_f denotes the final basis size so that $k_s < k_f$ where each is a multiple of b, and k_{con} denotes the number of eigenvalues that have converged. For the moment, we will assume $k_{con} = 0$. In our implementation, we begin by using Algorithm 3.1.3 to compute a block Arnoldi factorization

$$AQ(:, 1 : k_s) = Q(:, 1 : k_s + b)\widetilde{H}(1 : k_s + b, 1 : k_s)$$

where $Q \in \mathbb{C}^{n\times(k_s+b)}$ has orthonormal columns and a compact WY representation as in Equation 2.5. We will discuss the importance of this first expansion using block Arnoldi in Chapter 4.

The Schur form of the Rayleigh quotient $\widetilde{H} \in \mathbb{C}^{(k_s+b)\times(k_s)}$ is computed next so that we have

$$\widetilde{H}(1:k_s, 1:k_s)U_s = U_sT_s$$

with $U_s \in \mathbb{C}^{k_s\times k_s}$ such that $U_s^H U_s = I$ and $T_s \in \mathbb{C}^{k_s\times k_s}$ is upper triangular. As before, we could instead compute the real Schur form in which T_s is upper block triangular and work completely in real arithmetic. We will discuss this further in Chapter 4. Updating our block Arnoldi decomposition we have the block Krylov-Schur factorization

$$AZ(:, 1:k_s) = Z(:, 1:k_s+b)S(1:k_s+b, 1:k_s), \tag{3.15}$$

where $Z(:, 1:k_s) = Q(:, 1:k_s)U_s$, $Z \in \mathbb{C}^{n\times(k_s+b)}$ has orthonormal columns, the Rayleigh quotient is upper triangular, $S(1:k_s, 1:k_s)$, with a full $r \times k_s$ block on the bottom as depicted in Figure 3.1. This ends the initialization phase of our block

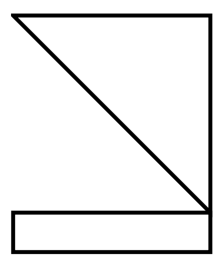


Figure 3.1: Block Krylov-Schur Decomposition

Krylov-Schur approach. Now we enter a cycle of expanding and contracting the Krylov decomposition until an eigenvalue may be deflated. The initial Krylov-Schur decomposition 3.15 is expanded in the same manner as in our block Arnoldi approach. Details may be found in Algorithm 3.1.3. The expanded Krylov decomposition is now

of the form

$$AQ(:, 1 : k_f + b) = Q(:, 1 : k_f + b)S(1 : k_f + b, 1 : k_f) \tag{3.16}$$

where $Q \in \mathbb{C}^{n\times(k_f+b)}$ has orthonormal columns and a compact WY representation, and the Rayleigh quotient has the structure depicted in Figure 3.2. If $k_{con} > 0$, constructing the compact WY representation of our expanded Z may involve a form as in Equation 3.7. We will address this in detail when we reach the end of a sweep.

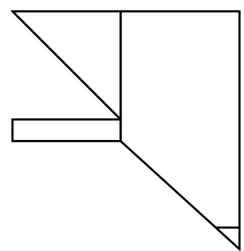


Figure 3.2: Expanded Block Krylov Decomposition

The next step is to compute the Schur form of the $k_f \times k_f$ principal submatrix in the upper left of S. Again, we use MATLAB's function **schur** to compute the upper triangular Schur factor. The Schur factor is then reordered as we did before in our block Arnoldi method. Algorithm 3.1.6 accomplishes this task. We then update the corresponding parts of S so that our decomposition has the form

$$AQ(:, 1 : k_f) = Q(:, 1 : k_f + b)S(1 : k_f + b, 1 : k_f) \tag{3.17}$$

where S again has the structure depicted in Figure 3.1. Once the desired eigenvalue approximations have been moved to the upper left of the matrix S, we check for convergence. Kressner [43] details the connection between the convergence criteria of ARPACK and possible extensions to a Krylov decomposition. Given a Krylov-Schur decomposition of order m,

$$AU_m = U_{m+1}\begin{bmatrix} B_m \\ b_{m+1}^T \end{bmatrix}$$

the direct extension of Equation 3.11 is given by

$$\|A(U_m w) - \lambda(U_m w)\|_2 = |b_{m+1}^T w| \leq \max\left\{\mathbf{u}\|B_m\|_F, tol \times |\lambda|\right\},$$

where $\|w\|_2 = 1$, $\mathbf{u}$ is the machine precision and tol is a user tolerance. Both of these criteria, the one of ARPACK and the Krylov extension, guarantee a small backward error. Kressner also discussed how the Krylov-Schur approach suggests a more restrictive convergence criteria based on Schur vectors rather than Ritz vectors. If no eigenvalues have been deflated, we may accept an approximation if

$$|b_1| \leq \max\left\{\mathbf{u}\|B_m\|_F, tol \times |\lambda|\right\}, \tag{3.18}$$

where b_1 is the first component of b_{m+1}^T. We discuss this in depth in Chapter 4. An extension of the convergence criterion based on Schur vectors to our block method is given by

$$\|S(k_f + 1 : k_f + b, 1)\|_2 \leq \max\left\{\mathbf{u}\|S(1 : k_f, 1 : k_f)\|_F,\ tol \times |\lambda|\right\} \tag{3.19}$$

where $\lambda = S(1,1)$, $\mathbf{u}$ is machine precision and tol is a user specified tolerance. Our block Krylov-Schur implementation can deflate converged eigenvalues one at a time, or n_{con} at a time where $1 \leq n_{con} \leq b$. If the eigenvalue satisfies Equation 3.19, we explicitly deflate by zeroing out the bottom $b \times 1$ block in the corresponding column of S and then truncate our Krylov-Schur decomposition. If no eigenvalues have converged, the Krylov-Schur decomposition is truncated. Truncation in either case is handled by Algorithm 3.1.10 and a compact WY representation of Z is computed. After truncation, the process begins again with expansion of the search subspace.

Algorithm 3.1.8: Block Krylov-Schur

Input: $A \in \mathbb{C}^{n \times n}$, $U \in \mathbb{C}^{n \times b}$ and number of desired eigenvalues k_{max}

Result: Partial Schur form given by $Z_{k_{max}}$ and $S_{k_{max}}$

Use Algorithm 3.1.3 to generate:

$AQ(:, 1 : k_s) = Q(:, 1 : k_s + b)\widetilde{H}(1 : k_s + b, 1 : k_s);$

Compute Schur form of Rayleigh quotient $\widetilde{H}(1 : k_s, 1 : k_s)$;

Update columns of Q and $\widetilde{H}(k_s + 1 : k_s + b, :)$;

Now we have a block Krylov-Schur decomposition:

$AZ(:, 1 : k_s) = Z(:, 1 : k_s + b)S(1 : k_s, 1 : k_s);$

while $k_{con} \leq k_{max}$ **do**

  Use Algorithm 3.1.9 to expand as in Arnoldi:

  $AZ(:, 1 : k_{con} + k_f) = Z(:, 1 : k_{con} + k_f + b)S(1 : k_{con} + k_f + r, 1 : k_{con} + k_f);$

  Compute the Schur form of the active part of the Rayleigh quotient:

  $S(k_{con} + 1 : k_{con} + k_f, k_{con} + 1 : k_{con} + k_f)U_s = U_s T_s;$

  Reorder Schur form using Algorithm 3.1.6: $T_s \leftarrow U_r T_s U_r$;

  Update corresponding pieces of Krylov decomposition:

  $Z(:, k_{con} + 1 : k_{con} + k_f) \leftarrow Z(:, k_{con} + 1 : k_{con} + k_f)U_r$

  $S(1 : k_{con}, k_{con} + 1 : k_{con} + k_f) \leftarrow S(1 : k_{con}, k_{con} + 1 : k_{con} + k_f)U_s U_r$

  $S(k_{con} + 1 : k_{con} + k_f + b, k_{con} + 1 : k_{con} + k_f + b) \leftarrow T_s$

  $S(k_{con} + k_f + 1 : k_{con} + k_f + b, :) \leftarrow S(k_{con} + k_f + 1 : k_{con} + k_f + b, :)U_s U_r;$

  if *converged* **then**

    Explicitly deflate n_{con} converged eigenvalues;

    $S(k_{con} + k_f + 1 : k_{con} + k_f + b, k_{con} + 1 : k_{con} + n_{con}) = \text{zeros}(b, n_{con})$

    $k_{con} = k_{con} + n_{con};$

    Truncate expansion to size k_s using Algorithm 3.1.10;

  else

    Truncate expansion to size k_s using Algorithm 3.1.10;

Algorithm 3.1.9: Expansion of block Krylov decomposition

Input: A, $Z \in \mathbb{C}^{n\times(k_{con}+k_s+b)}$, $S \in \mathbb{C}^{(k_{con}+k_s+b)\times(k_{con}+k_s)}$

Result: $Z \in \mathbb{C}^{n\times(k_{con}+k_f+b)}$, $S \in \mathbb{C}^{(k_{con}+k_f+b)}$

for $j = b_s : b_f$ **do**

Compute components for compact WY form of QR factorization:

$V = \mathrm{qr}(U(k_{con} + jb + 1 : n, 1 : b), 0)$; and compute scalars $\tau(k_{con} + \{j + 1\})$

for the elementary reflectors;

Update S: $S(1 : jb + k_{con}, k_{con} + \{j\}) = U(1 : jb + k_{con}, :)$

$S(k_{con} + \{j + 1\}, k_{con} + \{j\}) = \mathrm{triu}(V(1 : b, 1 : b))$;

Store reflectors in Y:

$Y(k_{con} + jb + 1 : n, k_{con} + \{j + 1\}) = \mathrm{eye}(\mathrm{size}(V)) + \mathrm{tril}(V, -1)$;

Build triangular factor for compact WY representation: $T = \mathrm{zlarft}(Y, \tau)$;

Explicitly build next block of Z:

$Z(:, k_{con} + \{j + 1\}) = Z(:, k_{con} + \{j + 1\}) - YTY^H Z(:, k_{con} + \{j + 1\})$;

If$j < b_2$ $U = AZ(:, k_{con} + \{j + 1\})$;

$U = U - YT^H Y^H U$;

$U(1 : k_{con} + k_s + b, :) = R^H U(1 : k_{con} + k_s + b, :)$;

Algorithm 3.1.10: Truncation of block Krylov decomposition

Input: S, Z and n_{con}

Result: truncated S and Z with compact WY form

if $n_{con} > 0$ **then**

  Explicitly deflate;

  for $i = 1 : n_{con}$ **do**

    $S(k_{con} + k_f + 1 : k_{con} + k_f + b, k_{con} + i) = \text{zeros}(b, 1);$

  $k_{con} = k_{con} + n_{con};$

  $Z(:, k_s + k_{con} + 1 : k_s + k_{con} + b) = Z(:, k_f + k_{con} - n_{con} + 1 : k_f + k_{con} - n_{con} + b);$

  $Z(:, k_s + k_{con} + r + 1 : k_{con} + k_f + b) = \text{zeros}(n, k_f - k_s);$

  $Z(k_{con} + k_s + b + 1 : n, k_s + k_{con} + b + 1 : k_{con} + k_f + b) =$ $\text{eye}(n - k_s - b - k_{con}, k_f - k_s);$

  $S = [S(1 : k_s + k_{con}, 1 : k_s + k_{con}); S(k_f + k_{con} - n_{con} + 1 :$ $k_{con} + k_f + b - n_{con}, 1 : k_s + k_{con})];$

  $n_{con} = 0;$

else

  $Z(:, k_s + k_{con} + 1 : k_s + k_{con} + b) = Z(:, k_f + k_{con} + 1 : k_f + k_{con} + b);$

  $Z(:, k_s + k_{con} + r + 1 : k_{con} + k_f + b) = \text{zeros}(n, k_f - k_s);$

  $Z(k_{con} + k_s + b + 1 : n, k_s + k_{con} + b + 1 : k_{con} + k_f + b) =$ $\text{eye}(n - k_s - b - k_{con}, k_f - k_s);$

  $S = [S(1 : k_s + k_{con}, 1 : k_s + k_{con}); S(k_{con} + k_f + 1 : k_{con} + k_f + b, 1 : k_s + k_{con})];$

Build new compact WY representation: $X = \text{qr}(Z(:, 1 : k_{con} + k_s + b), 0);$ and compute scalars $\tau(1 : k_{con} + k_s + b);$

$Y = \text{tril}(X, -1) + \text{eye}(n, k_{con} + k_s + b);$

$T = \text{zlarft}(Y, \tau); R = \text{triu}(X(1 : k_{con} + k_s + b, :));;$

3.1.4 Block Jacobi-Davidson

We now turn our focus from methods that project onto Krylov subspaces to an approach that uses a different projection technique. The Jacobi-Davidson approach was first proposed in 1996 by Sleijpen and Van der Vorst [58]. This method combined ideas from Davidson's work on large, sparse symmetric matrices from the 1970s [21] and Jacobi's iterative approaches to computing eigenvalues of symmetric matrices from the 1840s [37]. We present a brief discussion of Davidson's work and Jacobi's work to see the relationship between Krylov based methods such as Arnoldi and methods such as Jacobi-Davidson.

Jacobi introduced a combination of two iterative methods to compute the eigenvalues of a symmetric matrix. To see how Jacobi viewed eigenvalue problems, let A be an $n \times n$, diagonally dominant matrix with largest diagonal element $A(1,1) = \alpha$. An approximation of the largest eigenvalue and associated eigenvector is the Ritz pair (α, v) as in

$$A \begin{bmatrix} 1 \\ z \end{bmatrix} = \lambda \begin{bmatrix} 1 \\ z \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} \alpha & c^T \\ b & F \end{bmatrix} = \lambda \begin{bmatrix} 1 \\ z \end{bmatrix},$$

where $b, c \in \mathbb{R}^{n-1}$ and $F \in \mathbb{R}^{(n-1)\times(n-1)}$. Jacobi proposed to solve eigenvalue problems of this form using his Jacobi iteration. To see this, consider the alternative formulation of this system

$$\begin{aligned} \lambda &= \alpha + c^T z \\ (F - \lambda I)z &= -b. \end{aligned}$$

Jacobi solved the linear system on the second line using his Jacobi iteration by beginning with $z_1 = 0$ and getting an updated approximation θ_k for λ using a variation

of the iteration

$$\theta_k = \alpha + c^T z_k$$
$$(D - \theta_k I) z_{k+1} = (D - F) z_k - b,$$

where D is the diagonal matrix with the same diagonal entries as F. The first step was to make the matrix strongly diagonally dominant by applying rotations to precondition the matrix. Jacobi then proceeded with his iterative method that searched for the orthogonal complement to the initial approximation. Further details of Jacobi's work may be found in [58]. The key idea from his work is that all corrections came from the orthogonal complement of the initial approximation.

Davidson was also working with real symmetric matrices. Suppose we have a subspace V of dimension k, the projected matrix A has Ritz value θ_k and Ritz vector u_k, and that an orthogonal basis for V is given by $\{v_1, \ldots, v_k\}$. A measure of the quality of our approximation is given by the residual

$$r_k = A u_k - \theta_k u_k.$$

Davidson was concerned with how to expand the subspace V to improve the approximation and update u_k. His answer consisted of the following steps:

1. Compute t from the system $(D - \theta_k I)t = r_k$

2. Orthogonalize t against the basis for V: $t \perp \{v_1, \ldots, v_k\}$

3. Expand the subspace V by taking $v_{k+1} = t$

where D is a diagonal matrix with the same diagonal entries as A. The Jacobi-Davidson method combines elements from both approaches. Given an approximation u_k, the correction to this approximation is found in the projection onto the orthogonal complement of the current approximation. The projected matrix is given by

$$B = (I - u_k u_k^T) A (I - u_k u_k^T) \tag{3.20}$$

and rearranging terms yields

$$A = B + Au_k u_k^T + u_k u_k^T A - \theta_k u_k u_k^T,$$

where $\theta_k = u_k^T A u_k$. For a desired eigenvalue of A, say λ, that is close to θ_k, the desired correction t is such that

$$A(u_k + t) = \lambda(u_k + t), \text{ and } t \perp u_k. \tag{3.21}$$

Substituting Equation 3.20 into the desired correction Equation 3.21, and using some orthogonality relations, we have the following equation for the correction

$$(B - \lambda I)t = -r_k \tag{3.22}$$

where $r_k = Au_k - \theta_k u_k$ is the residual. Different approaches to solving the correction equation 3.22 result in different methods. If the solution is approximated by the residual, that is $t = r_k$, the correction is formally the same as that generated by Arnoldi. In the symmetric case, if $t = (D - \theta_k I)^{-1} r_k$, then we recover the approach proposed by Davidson. In the general case, combining the strategies of Jacobi and Davidson, the correction equation has the form

$$(B - \theta_k I)t = -r_k \text{ with } t \perp u_k, \tag{3.23}$$

where $(B - \lambda I)$ is replaced by $(B - \theta_k I)$. Approximating a solution to Equation 3.23 has been studied extensively since it was proposed. We will highlight recent developments later. The work by Sleijpen and Van der Vorst [58] set the foundation for the formulation of a Jacobi-Davidson style QR algorithm, JDQR, presented by Fokkema et al. [24], that iteratively constructs a partial Schur form. In [24], algorithms were presented for both standard eigenvalue problems and generalized eigenvalue problems, including the use of preconditioning for the correction equation and restart strategies. We will restrict our discussion to standard eigenvalue problems.

For the standard eigenvalue problem, the Jacobi-Davidson method picks an approximate eigenvector from a search subspace that is expanded at each step. If the search subspace is given by span$\{V\}$, then the projected problem is given by

$$(V^H AV - \theta V^H V)u = 0, \tag{3.24}$$

where $V \in \mathbb{C}^{n \times j}$ and in exact arithmetic $V^H V = I$. Equation 3.24 is solved yielding Ritz value θ, Ritz vector $q = Vu$ and residual vector $r = (A - \theta I)q$ where $r \perp q$. The projected eigenproblem 3.24 is reduced to Schur form by the QR algorithm. Fokkema et al. defined the $j \times j$ interaction matrix $M = V^H AV$ with Schur decomposition $MU = US$ where S is upper triangular and ordered such that

$$|S(1,1) - \tau| \leq |S(2,2) - \tau| \leq \ldots \leq |S(j,j) - \tau|,$$

where τ is some specified target. A Ritz approximation to the projected problem with Ritz value closest to τ is given by

$$(q, \theta) \equiv (VU(:,1), S(1,1)).$$

Additionally, useful information for the i eigenvalues closest to τ may be found in the span of the columns of $VU(:, 1 : i)$ with $i < j$. This facilitates a restart strategy by taking $V = VU(:, 1 : j_{min})$ for some $j_{min} < j$. Fokkema et al. [24] referred to this as an implicit restart. The second component of this approach determines how to expand the search subspace using iterative approaches due to Jacobi. The correction equation given by

$$(I - qq^H)(A - \theta I)(I - qq^H)v = -r \text{ with } q^H v = 0, \tag{3.25}$$

is solved approximately and the expanded search subspace becomes span$\{V, v\}$. The JDQR approach can compute several eigenpairs by constructing a partial Schur form. Both deflation and a restarting strategy are used. An interesting feature of JDQR is the option of preconditioning the correction equation 3.25. Due to the projections

involved, preconditioning is not straightforward. Detailed pseudocodes may be found in [24] and a MATLAB implementation **jdqr** is publicly available.

Since its introduction in 1996 by Sleijpen and Van der Vorst [58], much work has been devoted to understanding and improving the Jacobi-Davidson approach, especially in the case of symmetric and Hermitian matrices. Here we survey the major highlights that pertain to our block variant in the context of the NEP. There is a wealth of information available on the Jacobi-Davidson approach and a good starting point is the Jacobi-Davidson Gateway [33] maintained by Hochstenbach. As mentioned earlier, Fokkema et al. [24] discussed deflation and implicit restarts. Deflation and the search for multiple eigenvalues of Hermitian matrices was also studied by Stathopoulos and McCombs [61]. The connection between the inner and outer iterations was studied by Hochstenbach et al. [34]. This is a critical component of the Jacobi-Davidson approach as solving the correction equation too accurately at the wrong time can lead to the search subspace being expanded in ineffective ways. We will examine this issue when presenting our numerical results. Hochstenbach et al. proved a relation between the residual norm of the inner linear system and the residual of the eigenvalue problem. This analysis suggested new stopping criteria that improved the overall performance of the method. We will employ some of their heuristics in our block approach.

A block variant for sparse symmetric eigenvalue problems was formulated by Geus [27]. For general matrices with inexpensive action, for example large and sparse matrices, Brandts [17] suggested a variant of blocked Jacobi-Davidson based on his Ritz-Galerkin Method with inexact Riccati expansion. This method has a Riccati correction equation, that depending on the quality of the approximate solution, reduces to a block Arnoldi approach or a block Jacobi-Davidson approach. In fact, the Riccati correction equation, when linearized, becomes the correction equation of Jacobi-Davidson. Brandts's method solves the Riccati equation exactly and the ex-

tra work is demonstrated to be negligible in the case of matrices with inexpensive action. Further investigation into subspace expansion from the solutions of generalized algebraic Riccati equations is the subject of future work. Parallelization has been investigated, but mainly in the context of generalized eigenvalue problems for large Hermitian matrices [52, 3] and most recently for quadratic eigenvalue problems [68].

Our block version of Jacobi-Davidson, Algorithm 3.1.11, is a straight forward extension of JDQR from [24]. In the publicly available MATLAB implementation, **jdqr**, Fokkema et al. used existing implementations of the QR algorithm such as **schur** to compute the Schur form of the projected problem. MGS was used for the the construction of an orthogonal basis for the search subspace. Rather than using MGS, we opt as before to base our algorithm on the use of Householder reflectors. Several linear solvers are available to approximately solve the correction equation 3.25. The implementation **jdqr** allows the user to specify various methods, but as our stopping criteria depends on the method used, we choose to employ the generalized minimal residual method (GMRES). We also formulate our approach to allow for preconditioning of the correction equation, though we do not suggest strategies for identifying effective preconditioners.

We now present one sweep of our block Jacobi-Davidson method detailed in Algorithm 3.1.11. Here b denotes the block size, j_{min} is the minimum dimension of the search subspace, j_{max} is the maximum dimension of the search subspace, k_{con} is the number of converged eigenvalues and k_{max} is the number of desired eigenvalues. In the ensuing steps, we use similar notation as Fokkema et al. introduced in [24] to help with the discussion. We let $Q \in \mathbb{C}^{n \times k_{con}}$ be the matrix of converged Schur vectors, $K \in \mathbb{C}^{n \times n}$ is the preconditioner for $(A - \xi I)$ for some fixed value of ξ and define the

Algorithm 3.1.11: Block Jacobi-Davidson

Input: $A \in \mathbb{C}^{n\times n}$, $U \in \mathbb{C}^{n\times b}$, k_{max}, j_{min}, and j_{max}

Result: $AQ = QR$ with $Q \in \mathbb{C}^{n\times k_{max}}$ and $R \in \mathbb{C}^{k_{max}\times k_{max}}$

$j = 0$;
while $k_{con} \leq k_{max}$ **do**
  if $j = 0$ **then**
    Initialize search subspace v using Algorithm 3.1.12;
  else
    Solve b correction equations approximately using Algorithm 3.1.13;
  Expand search subspace: $V = [V, v]$;
  if $j > 0$ **then**
    $[V, temp] = \mathrm{qr}(V, 0)$ and construct compact WY form for V;
  Expand interaction matrix: $M = V^H AV$;
  Compute Schur form: $MU = US$;
  Reorder the Schur form S using Algorithm 3.1.6;
  $j = j + b$; $found = 1$;
  while $found$ **do**
    Compute Ritz vectors: $q = VU(:, 1 : b)$;
    Precondition the Schur vectors: $y = K^{-1}q$;
    Compute b residual vectors and associated norms:
    for $i = 1 : b$ **do**
      $r(:, i) = Aq(:, i) - S(i, i)q(:, i)$; $\mathrm{nres}(i) = \|r(:, i)\|_2$;
    if $Converged$ **then**
      Deflate and restart using Algorithm 3.1.14
    else
      Implicit restart using Algorithm 3.1.14

following:

$$\widetilde{Q} \equiv [Q, q], \qquad \text{the matrix } Q \text{ expanded by approximate Schur vectors } q,$$

$$\widetilde{Y} \equiv K^{-1}\widetilde{Q}, \qquad \text{the matrix of preconditioned Schur vectors,}$$

$$\widetilde{H} \equiv \widetilde{Q}\widetilde{Y}, \qquad \text{the preconditioned projector } \widetilde{Q}^H K^{-1}\widetilde{Q}.$$

We begin with a block $U \in \mathbb{C}^{n\times b}$, and in the initial sweep orthogonalize this using Householder. That is, we have

$$[V, R] = \text{qr}(U, 0)$$

so that $V \in \mathbb{C}^{n\times b}$ is such that $V^H V = I_b$. As in **jdqr**, we also allow for initializing the search subspace using Arnoldi. Convergence may suffer in the single vector case when using Jacobi-Davidson beginning from the initial vector. That is, the correction equation may not immediately provide useful information for building a desirable search subspace. To adjust for this, often some type of subspace expansion is used to generate an initial search subspace that may have better approximations to work with. The first phase of **jdqr** computes an Arnoldi factorization of size $j_{min}+1$ using the supplied initial vector. Then j_{min} of the Arnoldi vectors are used as the initial subspace V in the initial sweep of the Jacobi-Davidson method. We experimentally verified that this slightly improves the speed of convergence and incorporate this approach into our block algorithm. We note that a nice analysis of the correction equation is provided by Brandts [17]. If desired, we use Algorithm 3.1.3 and the starting block U to build a size j_{min} block Arnoldi decomposition and let $V \in \mathbb{C}^{n\times j_{min}}$ be the first j_{min} basis vectors. If this is not the initial sweep, $V \in \mathbb{C}^{n\times j_{min}}$ is our restarted search subspace and has orthonormal columns. In either case a compact WY representation of the search subspace V is constructed. The interaction matrix is computed by

$$M = V^H AV$$

and then the Schur form of M is computed using MATLAB's function **schur** so that $MU = US$. Next the Schur form is reordered using our earlier approach Algorithm 3.1.6 so that the diagonal entries of S are arranged with those closest to some target in the upper left and U is updated as well. The first b diagonal elements of S are the Ritz values with associated Ritz vectors given by

$$q = VU(:, 1 : b),$$

and are used to compute the b residuals

$$r(:, i) = Aq(:, i) - S(i, i)q(:, i) \qquad i = 1, \ldots, b \tag{3.26}$$

along with the corresponding norms of each residual $\|r(:, i)\|_2$. Next we check for convergence. In **jdqr**, a Ritz approximation is accepted if the norm of the residual is below a certain user specified tolerance with default value 1e-8. We use the same convergence criterion for individual Ritz approximations, but check to see if any of the b approximations has converged. If the Ritz pair satisfies our convergence criterion, then we explicitly deflate the converged eigenvalue and lock the approximation as detailed in Algorithm 3.1.14. If the approximation is not yet satisfactory, we move

---

Algorithm 3.1.12: Subspace Initialization

Input: $U \in \mathbb{C}^{n \times b}$ and j_{min}

Result: Initial search subspace v with compact WY representation

if *Starting with Arnoldi* **then**

  Construct size j_{min} block Arnoldi decomposition:

  $AU(:, 1 : j_{min}) = U(:, 1 : j_{min} + b)H(1 : j_{min} + b, 1 : j_{min})$

  with compact WY form of U using Algorithm 3.1.3;

  $v = U(:, 1 : j_{min})$;

else

  $[v, R] = \text{qr}(U, 0)$ and along with compact WY form of v;

Algorithm 3.1.13: Approximate Solution to Correction Equations

Update residuals:

$r = K^{-1}r$; $r = r - \widetilde{Y}\widetilde{H}^{-1}\widetilde{Q}^H r$;

for $i = 1 : b$ **do**

  Approximately solve the b correction equations of the form:

  $(I - \widetilde{Y}\widetilde{H}^{-1}\widetilde{Q}^H)(K^{-1}A - S(i,i)K^{-1})(I - \widetilde{Y}\widetilde{H}^{-1}\widetilde{Q}^H)z(:,i) = -r(:,i)$;

  using GMRES and orthogonalize against $\widetilde{Q}$;

to the inner iteration. Here b correction equations of the form 3.25 are solved to expand the search subspace. As only an approximate solution to each system is required, iterative methods for linear systems are the natural choice. We opted to use GMRES to solve each of the b correction equations. Rather than use MATLAB's function **gmres**, we opted to implement our own version of GMRES as we needed more control during the inner solve to compute the components required for our convergence criteria. After computing the b approximate solutions, we find ourselves back at the beginning of a sweep. We then continue the cycle of outer projections and inner linear solves increasing the dimension of our search subspace a block at a time. If the size of the search subspace has reached the maximum allowed dimension j_{max}, we restart as detailed in Algorithm 3.1.14.

One of the central issues in the Jacobi-Davidson approach is the connection between the outer projection and the inner linear system. As mentioned earlier, Hochstenbach and Notay [34] provide an in-depth analysis of how progress in the outer iteration is connected to the residual norm of the correction equation. We present their results for the standard eigenvalue to detail the stopping criteria we used in our implementation. Our work uses a subset of the heuristics provided by Hochstenback and Notay. The correction equation in the standard eigenvalue problem has the form 3.25 with residual vector $r = (A - \theta I)q$ and Ritz value $\theta = q^H Aq$.

Algorithm 3.1.14: Deflation and Implicit Restart

```
if Converged then
    if k_con = 0 then
        R = S(1,1);
    else
        R = [triu(R), Q(:,1:k_con)^H r_save(:,1); zeros(1,k_con), S(1,1)];
    k_con = k_con + 1;
    Y = Ỹ(:,1:k_con);
    H = H̃(1:k_con,1:k_con);
    V = VU(:,2:j);
    S = S(2:j,2:j);
    M = S;
    U = eye(j − 1);
    j = j − 1;
else
    Implicit restart;
    j = j_min;
    V = VU(:,1:j_min);
    S = S(1:j_min,1:j_min);
    M = S;
    U = eye(j_min);
```

Let the inner residual of the linear system be given by

$$r_{in} = -r - (I - qq^H)(A - \tau I)v. \tag{3.27}$$

then

$$r_{eig} = \min_{\xi} \frac{\|(A - \xi I)(q + v)\|}{\|q + v\|} \tag{3.28}$$

satisfies

$$\frac{|g - \beta s|}{1 + s^2} \leq r_{eig} \leq \begin{cases} \frac{\sqrt{g^2+\beta^2}}{\sqrt{1+s^2}} & \text{if } \beta < gs \\ \frac{g+\beta s}{1+s^2} & \text{otherwise,} \end{cases} \tag{3.29}$$

where $g = \|r_{in}\|$, $s = \|v\|$, and $\beta = |\theta - \tau + q^H(A - \tau I)v|$. Hochstenbach and Notay suggested exiting the inner iteration if at least one of the following holds:

- $g_k \leq \tau_1 \|r\|$ and $r_{eig} \leq \epsilon$
- $g_k \leq \tau_1 \|r\|$ and $\frac{\beta s}{1+s^2} > \frac{\epsilon}{2}$ and $g_k < \tau_3 \frac{\beta s}{\sqrt{1+s^2}}$
- $g_k \leq \tau_1 \|r\|$ and $\frac{\beta s}{1+s^2} > \frac{\epsilon}{2}$ and $k > 1$ and $\left(\frac{g_k}{g_{k-1}}\right)^2 > \left(2 - \left(\frac{g_k}{g_{k-1}}\right)^2\right)^{-1}$

where $\tau_1 = 10^{-1/2}$, $\tau_3 = 15$, a norm-minimizing method such as GMRES is used, and ϵ is the desired value of the residual norm. For the outer eigenvalue estimate, they show that one may use the estimate

$$r_{eig} \approx \sqrt{\left(\frac{\|r_{in}\|}{\sqrt{1+s^2}}\right)^2 + \left(\frac{\beta s}{1+s^2}\right)^2}. \tag{3.30}$$

As we use GMRES for the solution of the correction equations, we only report the results pertaining to the use of GMRES in the context of the standard eigenvalue problem. Extensive details on how to proceed when using alternatives to GMRES may be found in [34]. Their approach computed these values twice during the solution of the inner system, first when $\|r_{in}\| \leq \tau_1 \|r\|$ and then when $\|r_{in}\| \leq \tau_2 \|r\|$ where $\tau_2 = 10^{-1}$. Their heuristic choices of thresholds τ_1, τ_2, and τ_3 were validated with a selection of test problems but they suggest experimenting with other values and also suggest the last criterion may be optional. For the standard eigenvalue problem with GMRES used for the inner solve, r_{eig} is about $\frac{\|r_{in}\|}{\sqrt{1+s^2}}$ until r_{eig} reaches its asymptotic value $\frac{\beta s}{1+s^2}$ and further reduction of $\|r_{in}\|$ is useless.

The main numerical result reported for the standard NEP showed that the number of matrix-vector products remained about the same when using **jdqr** with the new stopping criteria versus the default settings, but the revised stopping criteria

increased the number of inner iterations which are less costly than the outer iterations. Hochstenback and Notay concluded that **jdqr** would perform better with their stopping criteria. We will explore the behavior of different stopping criteria in our numerical experiments. Dedicating the appropriate amount of work to solving the inner linear system is increasingly important in the case of multiple correction equations. The stopping criteria in our block Jacobi-Davidson approach consists of the first two suggestions with threshold parameters mentioned above and parameters consistent with using GMRES for the the linear solve. The situation becomes more complicated when solving b correction equations and we do not claim to have the optimal criteria. A detailed analysis is the subject of future work.

Before we present numerical results, we pause to summarize the methods and associated software that will be used in our comparison. Table 3.1 lists all the approaches used in the ensuing section. Each of the approaches listed in Table 3.1 has several paramters the user can control. MATLAB's **eigs** allows one to set the initial vector, the tolerance for the convergence criteria, the number of vectors in the search subspace, the number of desired eigenvectors, and the targeted subset of the spectrum. Baglama's **ahbeigs** uses many of the same parameters, but has a few more options. The user may set the block size and the number of blocks. One of the parameters unique to **ahbeigs** is the *adjust* parameter that adds additional vectors to the restart vectors after Schur vectors converge. This is intended to help speed-up convergence. Sleijpen's **jdqr** has several input parameters the user may set to control the calculation. One may set the tolerance for the stopping criteria, the minimum and maximum dimensions of the search subspace, the initial vector, the type of linear solver for the correction equation, the tolerance for the residual reduction of the linear solver, the maximum number of iterations for the linear solver, and whether or not to use a supplied preconditioner.

Table 3.1: Software for comparison of iterative methods

Name	Source	Description	Blocked
eigs	MATLAB	Built-in function based on ARPACK	No
ahbeigs	Baglama's website	MATLAB implementation of ABHA	Yes
jdqr	Sleijpen's website	MATLAB implementation of JDQR	No
bA	Our home-grown code	Explicitly restarted Arnoldi	Yes
bKS	Our home-grown code	Block Krylov-Schur	Yes
bjdqr	Our home-grown code	Block extension of JDQR	Yes
Jia	Not available	Block Arnoldi	Yes
bIRAM	Not available	bIRAM by Lehoucq and Maschhoff	Yes
Möller(L)	Not availale	bIRAM by Möller	Yes
MMöller(S)	Not available	bIRAM by Möller	Yes

Our implementation **bA** uses parameters such as the dimension of the search subspace, a target for the desired subset of the spectrum, a tolerance for the convergence criteria and the number of desired eigenvalues. Our implementation **bKS** has these same input parameters, but requires both the dimension of the contracted search subspace and the dimension of the expanded search subspace. Our implementation **bjdqr** is a block extension of **jdqr** but only uses GMRES as the linear solver. The parameters include the number of desired eigenvalues, the starting block of vectors, the minimum and maximum dimensions of the search subspace, and a tolerance for the stopping criteria.

3.2 Numerical Results

In this section we present numerical experiments to assess the performance of our block codes. We compare our blocked implementations to unblocked versions and to

Table 3.2: Ten eigenvalues of CK656 with largest real part

$\lambda_{1,2} =$	$5.5024, 5.5024$
$\lambda_{3,4} =$	$1.5940, 1.5940$
$\lambda_{5,6} =$	$1.4190, 1.4190$
$\lambda_{7,8} =$	$1.4120, 1.4120$
$\lambda_{9,10} =$	$1.1980, 1.1980$

alternate approaches that are publicly available. The purpose of these experiments is to get a good impression of how these methods actually perform in the context of the NEP. We hope to explore why block methods may be an attractive option, whether these block methods can handle difficult computations, and further understand the performance of the chosen methods. We will study how block size affects convergence and explore reasonable conditions for the underlying search subspaces. Each method from Section 3.1 has specific parameters that may affect performance, and we endeavor to understand as much as possible. All the ensuing Matlab comparisons are performed on a Mac Pro with dual 2.4 GHz Quad-Core Intel Xeon CPU and 8GB RAM running Mac OS X Version 10.8.4 with Matlab R2013a.

We begin with assessing one of the theoretical advantages of block methods, the computation of clustered or multiple eigenvalues. To explore this we selected a suitable matrix from the NEP Collection in Matrix Market. We chose CK656 which is a 656×656 real, nonsymmetric matrix with eigenvalues that occur in clusters of order 4 with each cluster consisting of two pairs of very nearly multiple eigenvalues. There is no information on the application of origin. For each approach, we attempt to compute the ten eigenvalues with largest real part which are given in Table 3.2. We fix the number of vectors in the search subspace and vary the block size b and the number of blocks n_b accordingly for the blocked versions. This makes comparisons

between our block Arnoldi method and our block Krylov-Schur method relatively easy as both approaches solve the same size projected eigenvalue problem at each iteration. Comparisons to Jacobi-Davidson based approaches are more difficult as the size of the projected eigenvalue problem grows at each step in the outer iteration, and there is the inner iteration to consider as well. It is important to note that an outer iteration of Jacobi-Davidson requires more overall work than an inner iteration and that both (inner iterations and outer iterations) involve multiplications by the full matrix. To understand the performance of our Jacobi-Davidson based approaches, we will conduct experiments with various dimensions of the search subspace. In Table 3.3, we report the block size, number of blocks in the search subspaces, the number of iterations, both inner and outer for our block Jacobi-Davidson, the total number of matrix-vector products (MVPs), the total number of block matrix-vector products (BMVPs), and the relative error of the resulting partial Schur decompositions. We do not report the level of orthogonality as most methods are based on Householder and loss of orthogonality was not observed to be an issue.

The initial 656×4 block of vectors was generated by MATLAB's **randn** function with state 7 and the appropriate number of columns of vectors used in each case presented in Table 3.3. The tolerance, the value of **tol** supplied by the user for all methods, was 1e-12. This was to ensure that all methods use the same input parameters, but we note that different stopping criteria are used for different implementations. As detailed in Chapter 4, the stopping criteria used in **bKS** is based on Schur vectors rather than Ritz vectors as in done in ARPACK. This is one of the challenges of comparing algorithms using software. Details of the role of **tol** may be found in each section discussing our implementations, in the user guide for ARPACK [47], and in the documentation for **ahbeigs** and **jdqr**.

We attempted several experiments with higher tolerances, e.g. 1e-6 and 1e-9, but **eigs** and unblocked **bA** occasionally missed a desired eigenvalue. It is worth noting

that the block methods did not experience the same difficulty. For all methods the search subspace was fixed at 20 vectors with the exception of a few additional computations for Jacobi-Davidson based methods in which we extended this to 40 vectors. We set the parameters in each method accordingly with a few exceptions. For **ahbeigs**, we did not use the default value for the parameter *adjust*. This parameter adjusts the number of vectors added when vectors start to converge to help speed up convergence. The default value is *adjust*=3 and the sum of the number of desired eigenvalues and the parameter *adjust* should be a multiple of the block size. We attempted this experiment with *adjust*= 0, and then again with the default setting. The performance was noticeably different and we will elaborate on this momentarily. We also note that the documentation of **ahbeigs** recommends ten or more blocks for the approach to converge and compute all desired eigenvalues. The size of the truncated subspace in the block Krylov-Schur approaches was fixed at 8. We experimented a bit with some of the options for **jdqr**, but ended up using the default values of most parameters. These include the use of GMRES with a tolerance of 0.7^j for the residual reduction and a maximum of five iterations for the linear solve. We did not use a preconditioner for $A - \theta I$ in the the correction equation and we point out that the Jacobi-Davidson based methods benefit greatly when a good preconditioner is available.

As displayed in Table 3.3, every approach successfully found the desired eigenvalues in this experiment. The first three results from **ahbeigs** forced the method to use no additional vectors. This did not seem to affect the performance for computations using 10 or more blocks, but the performance was dramatically different for blocks of size four. Over 6,000 MVPs were required compared to only 320 when additional vectors were allowed, but we were using only half of the recommended number of blocks. The worst performance in terms of total number of MVPs was observed by **bA**. This was not entirely unexpected. In the unblocked case, comparing **bA** to

Table 3.3: Computing 10 eigenvalues for CK656

Method	b	n_b	Iterations	MVPs (BMVPs)	$\frac{\|AZ-ZS\|_2}{\|A\|_2}$
eigs	1	20	–	111	9.716e-13
ahbeigs	1	20	–	117	9.262e-13
ahbeigs	2	10	–	144 (72)	6.959e-13
ahbeigs	4	5	–	6476 (1619)	1.005e-12
ahbeigs	1	20	–	107	9.262e-13
ahbeigs	2	10	–	130 (65)	6.959e-13
ahbeigs	4	5	–	320 (80)	1.005e-12
bA	1	20	21	430	4.372e-13
bA	2	10	29	600 (300)	1.212e-13
bA	4	5	40	840 (210)	1.774e-13
bKS	1	8, 20	11	140	5.606e-14
bKS	2	4, 10	12	152 (76)	1.858e-13
bKS	4	2, 5	23	284 (71)	1.012e-13
bjdqr	1	8, 20	70, 275	354	1.888e-13
bjdqr	2	4, 10	39, 299	385 (39)	1.598e-13
bjdqr	4	2, 5	25, 376	484 (25)	1.332e-13
bjdqr	1	16, 40	60, 231	307	1.614e-13
bjdqr	2	8, 20	31, 240	318 (31)	1.488e-13
bjdqr	4	4, 10	21, 317	417 (21)	1.339e-13
jdqr	1	8, 20	98, –	351	1.025e-13
jdqr	1	16, 40	82, –	292	1.371e-13

eigs is comparing Arnoldi with explicit restart to IRAM. It was expected that IRAM would perform better as it employs a much better restart strategy. Both **ahbeigs** and our **bKS** required comparable total number of MVPs for the runs in which additional vectors were allowed for **ahbeigs**. In terms of iterations, **bKS** required nearly the same for blocks of size one and two and performed nearly the same number of total MVPs.

The story for the Jacobi-Davidson based methods is harder to tell. In Table 3.3, outer and inner iterations are reported for **bjdqr** but only outer iterations are available for **jdqr**. The default stopping criteria for the correction equation in **jdqr** was used. Our unblocked **bjdqr** performed about the same number of matrix-vector products as **jdqr**, but invested more in the solution to the correction equation. This was somewhat anticipated as Hochstenbach and Notay [34] experienced the same result when using this stopping criteria and **jdqr**. This result also seems to suggest that the work invested in a more refined stopping criteria may be worthwhile as less outer iterations will result in better overall performance as the inner iterations require less work. We verified the number of MVPs for **jdqr** by tracking the number of times the function providing the matrix was accessed as we did for **eigs** and **ahbeigs**. Overall, the total number of MVPs remained relatively consistent for computations with our **bjdqr**, though **ahbeigs** and **bKS** required less on average. The Jacobi-Davidson based approaches seem to benefit more from a larger search subspace, specifically in the case where more vectors are used for the implicit restart. We will further explore the role of the dimension of the search subspace in additional numerical experiments.

Next we repeat the same experiment but increase the dimension of the search subspace. We set the number of vectors to be 48 with 20 used in the contracted subspace for **bKS**. For the Jacobi-Davidson based approaches, we set $j_{min} = 36$ and $j_{max} = 60$. The results are reported in Table 3.4. Increasing the subspace had some very interesting results. For our **bA** approach with block size $b = 4$,

Table 3.4: Computing 10 eigenvalues for CK656, expanded search subspace

Method	b	n_b	Iterations	MVPs (BMVPs)	$\frac{\|AZ-ZS\|_2}{\|A\|_2}$
eigs	1	48	–	112	9.716e-13
ahbeigs	1	48	–	118	2.024e-14
ahbeigs	2	24	–	148 (74)	1.302e-14
ahbeigs	4	12	–	252 (63)	1.754e-14
ahbeigs	1	48	–	116	2.370e-14
ahbeigs	2	24	–	144 (72)	1.373e-14
ahbeigs	4	12	–	200 (50)	3.976e-13
bA	1	48	11	538	3.928e-15
bA	4	12	23	332 (166)	5.796e-14
bKS	1	20, 48	10	300	5.309e-15
bKS	2	10, 24	10	300 (150)	3.037e-15
bKS	4	5, 12	21	328 (82)	3.705e-15
bjdqr	1	36, 60	40, 156	232	1.560e-14
bjdqr	2	18, 30	23, 176	258 (23)	1.170e-13
bjdqr	4	9, 15	18, 272	380 (18)	1.586e-13
bjdqr	1	72, 120	18, 68	158	3.027e-14
bjdqr	2	36, 60	9, 64	154 (9)	2.968e-14
bjdqr	4	9, 15	12, 176	296 (12)	1.345e-13
jdqr	1	36, 60	53, –	232	2.570e-14
jdqr	1	72, 120	14, –	129	3.798e-14

the required MVPs decreased dramatically from 840 to 332. The performance of **ahbeigs** also improved with the increased search subspace. As the number of blocks

was consistently larger than the recommended 10, the difference between the runs with and without the additional vectors was less significant. The additional vectors only seemed to be needed with block size $b = 4$ where the total number of MVPs was reduced by about 20%. The benefit of blocks was most evident in **bjdqr** as with the increased search subspace, slightly less total MVPs were required when comparing blocks of size one and blocks of size two. One of the most interesting results illustrated in these experiments is that **bKS** requires less total MVPs with a smaller search subspace. When restricting the search subspace to 20 vectors, **bKS** required half as many MVPs as it did for blocks of size 1 and 2. For blocks of size 4, **bKS** required about 12% less flops when working with a smaller search subspace. We note that the difference between the size of the expanded search subspace and the contracted search subspace increased from 12 to 28 vectors. The increased number of total MVPs required for **bKS** here seems to be tied to this increase. To verify this, we repeated this experiment just for **bKS** with the larger search subspace and only 12 additional vectors in the expansion phase, that is we set $k_s = 36$ rather than $k_s = 20$. The results are presented in Table 3.5. The results in Table 3.5 demonstrate that **bKS**

Table 3.5: Computing 10 eigenvalues for CK656, $k_s = 36$

Method	b	n_b	Iterations	MVPs (BMVPs)	$\frac{\|AZ-ZS\|_2}{\|A\|_2}$
bKS	1	36, 48	10	156	4.105e-15
bKS	2	18, 24	10	156 (78)	4.490e-15
bKS	4	9, 12	13	192 (48)	3.061e-12

performs similarly to the experiment with only 20 vectors in the search subspace. This seems to indicate that the difference between the dimension of the expanded search subspace and the dimension of the contracted search subspace is an important part of the overall performance of **bKS**. If the difference is too large, additional and

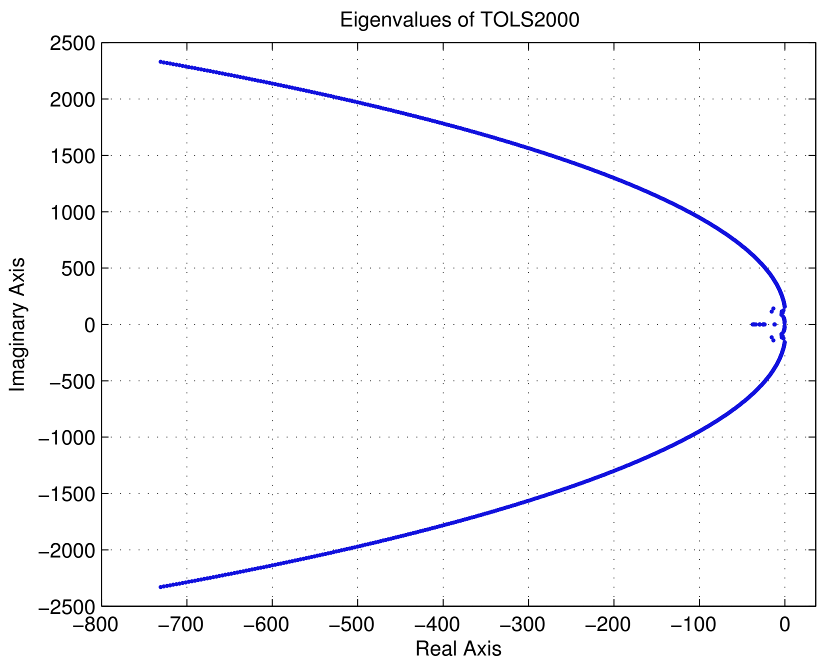


Figure 3.3: Complete Spectrum of TOLS2000

seemingly unnecessary work is performed. This is in contrast to **ahbeigs** and the Jacobi-Davidson based approaches which all perform better on average with a larger search subspace.

In our next experiment, we examine a matrix with difficult to compute eigenvalues. Here we hope to assess whether or not our implementations satisfies our hope for robustness. This example comes from the NEP Collection in Matrix Market as well. The matrix TOLS2000 is a Tolosa matrix from aerodynamics, related to the stability analysis of a model of a plane in flight. The eigenmodes of interest are complex eigenvalues with imaginary part in a certain frequency range determined by engineers. Figure 3.3 shows the complete spectrum. This computation aims to compute eigenvalues with largest imaginary part and the associated eigenvectors. The matrix is sparse and highly nonnormal making it potentially difficult to compute a few eigenpairs. Jia [38] computed the three eigenvalues with largest imaginary part

to be

$$\lambda_1 = -730.68859 + 2330.11977i,$$

$$\lambda_2 = -726.98657 + 2324.99172i,$$

$$\lambda_3 = -723.29395 + 2319.85901i,$$

using a block Arnoldi approach with refined approximate eigenvectors, and we will make indirect comparisons to his results as there is no publicly available code. We can make such a comparison thanks to efforts such as Matrix Market. Though the codes may not be available, the matrices used are available and we can compare to some extent to preious results. Jia observed the results in Table 3.6 with his proposed method. Jia's results show that his method benefited from a larger search subspace

Table 3.6: Summary of results presented by Jia [38]

b	n_b	Iterations	MVPs	$\lVert AZ - ZS \rVert_2$
2	25	67	3350	7.9e-7
2	30	33	1980	8.1e-7
2	35	26	1820	7.2e-7
2	40	32	2560	9.1e-7
2	50	11	1100	1.9e-7
3	20	88	5280	6.0e-7
3	30	20	1800	8.4e-7
3	40	8	960	6.3e-7

as the number of MVPs decreased when the number of blocks in the search subspace increased. This is similar to the behavior of **ahbeigs** in previous experiments. Here, we set the tolerance to 1e-9 to compare with the results by Jia and also by Baglama [7].

Again we experiment with allowing **ahbeigs** no additional vectors and the default setting. The initial block was generated using MATLAB's **randn** with state 7 as before with the appropriate number of columns used in each computation. Jia examined blocks of size two with 25 to 50 blocks and blocks of size three with 20 to 40 blocks and we selected various combinations to allow for a meaningful comparison. As was the case for the experiments performed by Jia, our block Arnoldi, **bA**, failed to converge for various block sizes with several different dimensions of the search subspace. Baglama found that **jdqr** failed to converge as well and we observed this for both **jdqr** and our **bjdqr** version with refined stopping criteria when working with blocks of size one. Even with blocks of various sizes, our **bjdqr** failed to converge. This could be due to the difficult nature of the problem and partly due to not using a preconditioner.

Both **eigs** and **ahbeigs** have options for computing the eigenvalues with largest imaginary part by setting the input option SIGMA='LI'. When attempting to locate the desired eigenvalues using the appropriate input string, both **eigs** and **abheigs** returned the three eigenvalues with largest imaginary part in magnitude. That is, both routines returned some combination the complex conjugates

$$\lambda_1 = -730.68859 + 2330.11977i,$$

$$\lambda_2 = -730.68859 - 2330.11977i,$$

$$\lambda_3 = -726.98660 \pm 2324.99171i,$$

rather than the approximations offered by Jia. This could be an issue specific to MATLAB as the documentation on **eigs** does not include how the interface to ARPACK is achieved. MATLAB's **eigs** can take a real or complex value as input for SIGMA, but **ahbeigs** only has the option of a real number or the appropriate string to designate the location and it works in only real arithmetic. We attempted to use various targets for **eigs** with no success. As Baglama [7] did not report the actual eigenvalues

computed, we report only the results we were able to generate using **ahbeigs**. In Table 3.7 we report the same information as before adding the number of successfully computed eigenvalues based on the ones reported by Jia, giving credit for complex conjugates.

The only approach to successfully compute all three desired eigenvalues was **bKS**. When compared to the results in Table 3.6, we see that **bKS** performed significantly less MVPs while using less vectors in the search subspace. For example, when using 10 blocks of size three **bKS** required 750 MVPs compared to Jia's approach that required 5280 MVPs when 20 blocks of size three were used. Increasing the number of blocks to 40 blocks brought the number of total MVPs required by Jia's approach much closer (down to 960) but this required four times the number of vectors in the search subspace.

Again we explored the effect of additional vectors for **ahbeigs**. In Table 3.7 there are select duplicate runs for **ahbeigs**. The first used no additional vectors and the second used the default value. Forcing the method to not use any additional vectors resulted in rather erratic behavior. For blocks of size two, additional vectors helped when the search subspace consisted of 15 blocks but performance suffered when using 30 blocks. Due to this, all ensuing runs were performed with the default setting. In all the results by **ahbeigs**, the best performance in terms of MVPs occurred when using 15 blocks of size two and additional vectors.

The results for **bKS** are mainly what one would expect. First, the configuration that required the least amount of total MVPs was unblocked **bKS**. This was followed closely by **bKS** using 15 blocks of size five and then 10 blocks of size three. Two different configurations of **bKS** outperformed **eigs** and a third required approximately the same number of total MVPs. Again, more MVPs were required for configurations of **bKS** when the difference between the expanded and contracted subspace was larger. The expanded search subspace needed to be large enough, but making it too

Table 3.7: Computing three eigenvalues with largest imaginary part for TOLS2000

Method	b	n_b	Iterations	k_{con}/k_d	MVPs (BMVPs)	$\|AZ - ZS\|_2$
eigs	1	30	–	2/3	746	8.305e-7
ahbeigs	1	30	–	2/3	1806	6.291e-7
ahbeigs	1	30	–	2/3	1580	1.107e-6
ahbeigs	2	15	–	2/3	1424 (712)	1.563e-6
ahbeigs	2	15	–	2/3	942 (471)	1.990e-6
ahbeigs	2	30	–	2/3	1740 (870)	1.604e-6
ahbeigs	2	30	–	2/3	2166 (1083)	3.518e-8
ahbeigs	3	10	–	2/3	2022 (674)	1.138e-6
ahbeigs	3	30	–	2/3	1350 (450)	1.969e-6
ahbeigs	5	6	–	2/3	3605 (721)	2.345e-6
ahbeigs	5	6	–	2/3	4090 (818)	2.251e-6
ahbeigs	5	10	–	2/3	1810 (362)	2.188e-6
ahbeigs	5	15	–	2/3	1960 (392)	2.020e-6
bKS	1	12, 30	27	3/3	498	4.541e-9
bKS	2	6, 15	64	3/3	1164 (582)	9.136e-9
bKS	3	4, 10	41	3/3	750 (250)	1.497e-6
bKS	2	12, 30	32	3/3	1176 (588)	1.497e-6
bKS	3	8, 20	33	3/3	1212 (404)	7.266e-9
bKS	5	4, 10	36	3/3	1100 (220)	4.348e-9
bKS	5	5, 10	38	3/3	975 (195)	1.007e-6
bKS	5	10, 15	23	3/3	625 (125)	9.105e-7
bKS	5	5, 15	70	3/3	3525 (705)	4.768e-9
bKS	10	2, 7	41	3/3	2070 (207)	1.231e-7
bKS	10	5, 10	28	3/3	1450 (145)	2.113e-6

large relative to the contracted subspace was not necessarily beneficial. The best performance was observed when a modest number of blocks were used to expand, specifically six blocks of size three and five blocks of size five. Overall, **bKS** seems flexible enough to work with a variety of configurations. We will further explore the relationships between block size, number of blocks (or dimension of search subspace), required iterations, and required matrix-vector products.

It is worth noting that for this numerical experiment we needed to deflate only one eigenvalue at a time in our successful approaches. Initial runs showed detection of more than one eigenvalue at a time, but the accuracy suffered. For most of the multiple deflations we observed representativity of the partial Schur form $\|AZ - ZS\| \approx \mathcal{O}(10^{-7})$. Our **bKS** typically looks for multiple deflations, but with the conditioning of this problem we needed to be a bit less ambitious to preserve accuracy. This experiment shows that our **bKS** approach performs well even with a challenging computation.

Next we consider a matrix used in experiments by Möller [49], by Lehoucq and Maschhoff [46], and by Baglama [7]. The purpose of this experiment is to make indirect comparisons to versions of bIRAM presented separately by Möller and by Lehoucq and Maschhoff as there are no publicly available codes. As Baglama [7] did, we refer to the two methods presented by Möller [49] as **Möller(S)** and **Möller(L)** and to the work by Lehoucq and Maschhoff as **bIRAM**. The matrix under consideration is HOR131 from the Harwell-Boeing Collection available on Matrix Market. The matrix is a 434×434 nonsymmetric matrix and comes from a flow network problem. We desire to compute the 8 eigenvalues with largest real part. We set the number of stored vectors to be 24, set the tolerance to be 1e-12, and generate the same initial starting block as we have in previous experiments. We opted to use the default for *adjust* in **ahbeigs**. We also verified the accuracy of the computed eigenvalues by comparing to the those computed by MATLAB's **eig** but we do not report those

Table 3.8: Summary of results for HOR131

Method	b	n_b	MVPs
bIRAM	1	24	77
bIRAM	2	12	84
bIRAM	3	8	99
bIRAM	4	6	108
Möller(S)	1	24	88
Möller(S)	2	12	136
Möller(S)	4	6	264
Möller(L)	1	24	79
Möller(L)	2	12	93
Möller(L)	4	6	105

details as all eigenvalues were computed within the desired tolerance. In Table 3.8 we report the results for indirect comparison.

In Table 3.9 we report the results of our computations. In Table 3.10 we present the results of our computations for Jacobi-Davidson based approaches. There are several things to note among the results for this experiment. First, **bA** is again the worst performer in terms of MVPs and again this was somewhat expected. Both **ahbeigs** and **bKS** seem to be competitive with **bIRAM**, **Möller(S)** and **Möller(L)**. The total number of MVPs required is similar and in our numerical experiments we have seen that different initial vectors can account for enough MVPs to consider these approaches as competitive. It is especially interesting to note that all the methods represented in Table 3.8 require more MVPs as the block size increases. This seems to be the case for **ahbeigs**, **bA** and **bjdqr**. The lone exception is **bKS**. As demonstrated

Table 3.9: Computing 8 eigenvalues with largest real part for HOR131, Krylov

Method	b	n_b	Iterations	MVPs (BMVPs)	$\|AZ - ZS\|_2$
eigs	1	24	–	83	2.958e-15
ahbeigs	1	24	–	83	2.096e-15
ahbeigs	2	12	–	140 (70)	2.295e-14
ahbeigs	3	8	–	180 (60)	2.100e-13
ahbeigs	4	6	–	316 (79)	1.787e-13
bA	1	24	17	416	1.627e-13
bA	2	12	20	496 (248)	6.815e-13
bA	3	8	24	600 (200)	5.410e-13
bA	4	6	25	632 (158)	7.866e-13
bKS	1	16, 24	10	96	1.301e-14
bKS	1	18, 24	11	84	5.991e-14
bKS	2	6, 12	11	144 (72)	2.172e-14
bKS	2	8, 12	12	112 (56)	6.173e-13
bKS	2	10, 12	16	84 (42)	3.430e-13
bKS	3	4, 8	13	168 (56)	1.396e-13
bKS	3	5, 8	14	141 (47)	1.396e-13
bKS	3	6, 8	13	114 (38)	1.396e-13
bKS	4	2, 6	16	264 (66)	6.044e-13
bKS	4	3, 6	14	180 (45)	2.175e-13
bKS	4	4, 6	19	168 (42)	2.738e-13

in Table 3.9, as we increase the dimension of the contracted subspace, the number of total MVPs decreases consistently for all block sizes reported in Table 3.9. This

Table 3.10: Computing 8 eigenvalues with largest real part for HOR131, JD

Method	b	n_b	Iterations	MVPs (BMVPs)	$\|AZ - ZS\|_2$
bjdqr	1	18, 30	37, 144	199	1.030e-12
bjdqr	2	9, 15	24, 184	250 (24)	4.382e-13
bjdqr	3	6, 10	18, 204	276 (18)	4.052e-13
bjdqr	4	4, 32	15, 224	300 (15)	1.775e-12
bjdqr	1	24, 48	26, 100	150	1.002e-12
bjdqr	2	12, 24	19, 144	206 (19)	1.136e-12
bjdqr	3	8, 16	16, 180	252 (16)	1.051e-12
bjdqr	4	6, 12	12, 176	248 (12)	1.471e-12
jdqr	1	18, 30	65, –	182	8.546e-15
jdqr	1	36, 60	14, –	67	1.954e-13

suggests that by finding the optimal configuration for the search subspace, **bKS** can be a very competitive approach.

The Jacobi-Davidson results in Table 3.10 tell much the same story as before for these types of methods. They all seem to require a bit more MVPs in general, but benefit from a larger search subspace. Increasing j_{min} and j_{max} makes **bjdqr** competitive but requires more vectors in the search subspace which then requires the solution of a larger eigenvalue problem than those methods not based on Jacobi-Davidson. Additional storage is also required.

Thus far, the performance of our **bKS** approach has been the most consistent among the iterative approaches discussed in this section. It has handled difficult computations and done so in a robust and efficient manner. To get a better understanding of how block size and dimension of the search subspace affect the performance, we

embark on some additional numerical experiments. The experiments performed to this point have focused on sparse matrices, from real applications, that others have used to assess performance of approaches to the NEP. We now consider dense random matrices. We begin with a comparison similar to the previous numerical experiments. We seek to compute the five eigenvalues with smallest real part for a $2,500 \times 2,500$ real matrix generated using MATLAB's **randn** with initial state 4. This populates the matrix with random numbers drawn from the standard normal distribution. We used the same initial starting block as in previous computations and again set the tolerance to 1e-12. The only other parameter we set is fixing 100 vectors in the search subspaces. For **ahbeigs** we use default values for the remaining parameters. We set the contracted search subspace dimension to $k_s = 72$ and adjusted the dimension of the expanded kf as close to 100 as possible using multiples of the block sizes. The results of our experiment are presented in Table 3.11. We observed that **bA** failed to converge for various configurations and **jdqr** had difficulties as well. We had to set the tolerance to 1e-11 to generate the results in Table 3.11 as it failed to converge with the tolerance set at 1e-12. The results in Table 3.11 show that **bKS** performs better than MATLAB's **eigs** and Baglama's **ahbeigs** for blocks up to size three. Larger blocks increase the total number of required MVPs, but not on the same scale as what is required by **ahbeigs**. Our Jacobi-Davidson implementation was able to locate all five desired eigenvalues where **jdqr** needed to relax the tolerance. This is most likely due to the difference in stopping criteria. As we have seen in previous experiments, we may be able to adjust the values of k_s and k_f to reduce the number of MVPs required and increase overall performance.

So far, **bKS** has performed well in difficult eigenvalue computations involving sparse matrices and demonstrated that it is an attractive option for computing a few eigenvalues of dense random matrices. We now turn our focus on only **bKS** as we attempt to understand how it behaves for dense random matrices. For the ensuing

Table 3.11: Computing 5 eigenvalues with smallest real part for random matrix

Method	b	n_b	Iterations	MVPs (BMVPs)	$\|AZ - ZS\|_2$
eigs	1	100	–	1687	5.987e-13
ahbeigs	1	100	–	1588	3.632e-13
ahbeigs	2	50	–	3502 (1751)	5.400e-13
ahbeigs	3	33	–	4869 (1623)	3.572e-13
ahbeigs	4	25	–	11188 (2797)	4.899e-13
ahbeigs	5	20	–	9010 (1802)	4.385e-13
ahbeigs	6	17	–	12522 (2087)	5.451e-13
bKS	1	72, 100	22	688	4.488e-13
bKS	2	36, 50	33	996 (498)	4.125e-13
bKS	3	24, 33	48	1368 (456)	3.204e-13
bKS	4	18, 25	65	1892 (473)	4.488e-13
bKS	6	12, 17	92	2832 (472)	5.119e-13
bjdqr	1	84, 108	371, 1463	1918	1.047e-14
bjdqr	2	42, 54	238, 1835	2395 (238)	9.896e-15
bjdqr	3	38, 36	175, 1988	2597 (175)	1.045e-14
bjdqr	4	21, 27	156, 2350	3058 (156)	1.471e-12
jdqr	1	84, 108	266, –	1631	3.858e-12

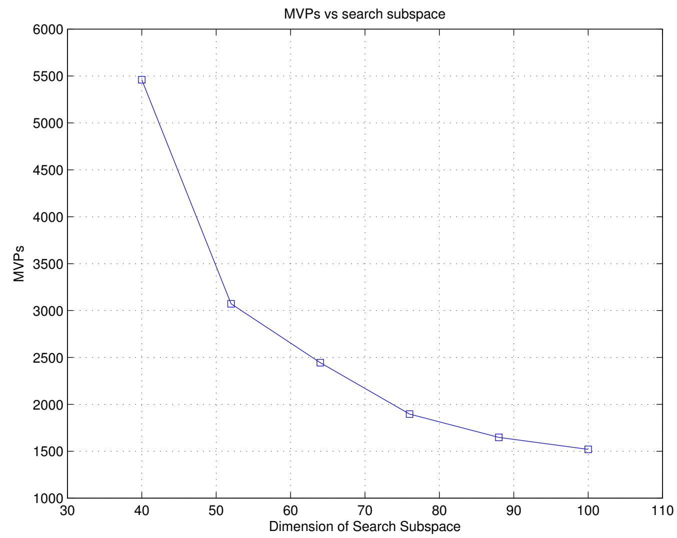


Figure 3.4: MVPs versus dimension of search subspace

experiment, we explore the effect of varying the dimension of the search subspace. Figure 3.4 displays the total number of required MVPs versus the dimension of the search subspace. Here we increased the search subspace from $k_s = 20$ and $k_f = 40$ to $k_s = 80$ and $k_f = 100$ adding 12 vectors each time. Initially we observed a sharp decrease in the total number of MVPs that leveled out. This suggests that care must be taken to make the search subspace large enough but not necessarily too large. The situation for the required number of iterations is nearly identical and is depicted in Figure 3.5. In Figure 3.6 we present the number of MVPs required to compute an increasing number of desired eigenvalues. Here we use a random $2,500 \times 2,500$ complex matrix generated by MATLAB again using state 4. The initial block was constructed in the same manner as before and the block size was fixed at 5. We computed up to 100 eigenvalues and noted the number of required MVPs. Here we see that the required work increases but the curve is slightly concave down.

Finally, we examine the relationship between the number of iterations and the problem size. We fix the block size at $b = 5$, the search subspace at $k_s = 40$ and

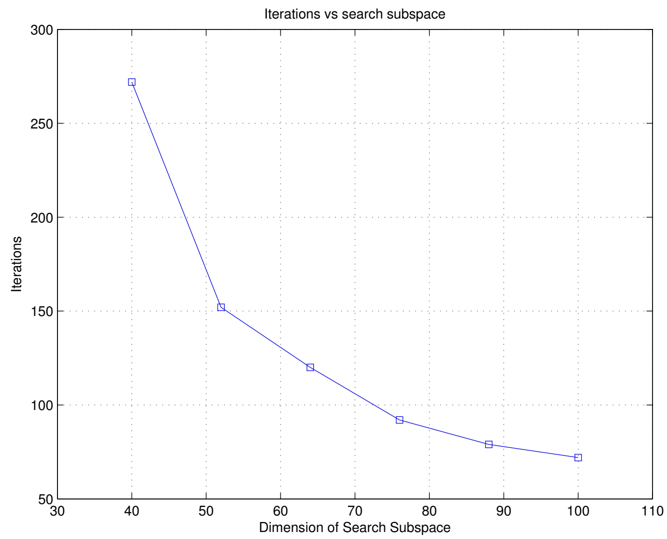


Figure 3.5: Iterations versus dimension of search subspace for block Krylov-Schur method.

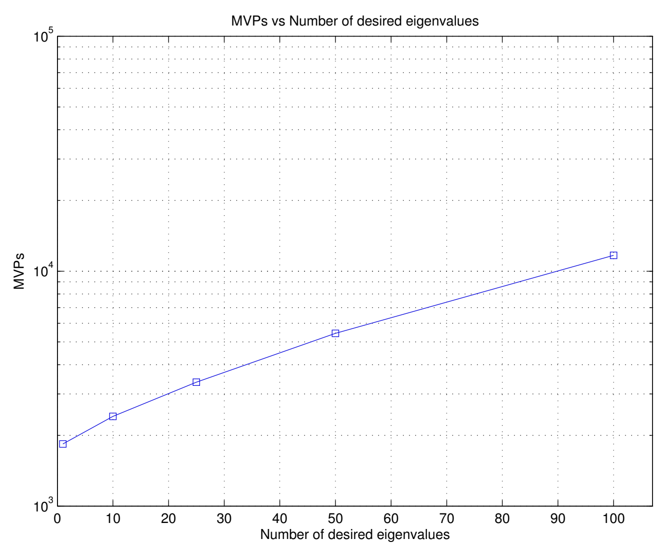


Figure 3.6: MVPs versus number of desired eigenvalues for block Krylov-Schur method.

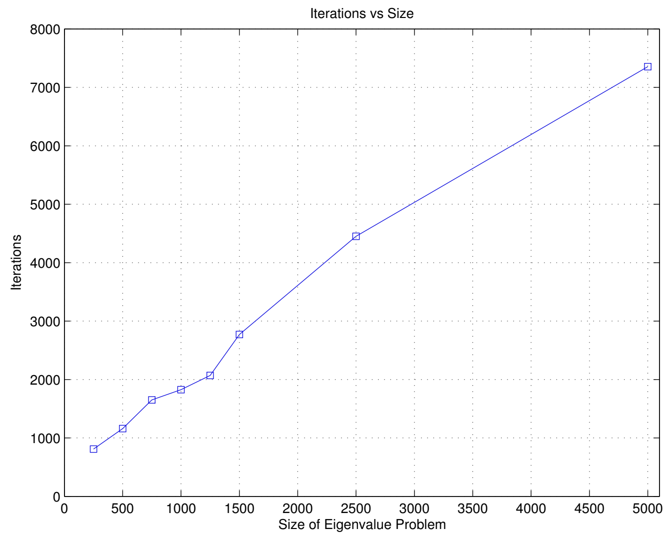


Figure 3.7: Iterations versus size of matrix for block Krylov-Schur with $b = 5$, $k_s = 40$ and $k_f = 75$.

$k_f = 75$, and now vary the problem size from a random 250×250 complex matrix to a random $5,000 \times 5,000$ matrix all constructed using MATLAB's **randn** with state 4. The iterations seem to grow linearly with the size of the matrix, but we point out this is for a fixed search subspace configuration. Previous numerical experiments suggest varying k_s, k_f and the block size can dramatically affect performance.

3.3 Conclusion and Future Work

In this Chapter, we have analyzed the performance of several iterative approaches to the NEP, including some novel formulations we implemented in MATLAB. Our numerical experiments indicate that our novel block Krylov-Schur with Householder orthogonalization compares well with current standards among iterative methods in the case of sparse nonsymmetric matrices. We were able to find configurations of the parameters for **bKS** that showed it to be competitive with IRAM in the MATLAB environment. Specifically, we found that we could adjust the block size, the dimension of the contracted search subspace, and the dimension of the expanded search subspace so that our block method performed approximately the same number of MVPs as the state-of-the-art serial implementation of IRAM provided by ARPACK. The

implementation **bKS** was shown to be robust and handled a variety of computations, including multiple and clustered eigenvalues, and challenging calculations involving highly nonnormal matrices. Additionally, our approach is able to compute any size partial Schur decomposition.

4. Block Krylov Schur with Householder

In this chapter, we present our current work on accelerating algorithms designed for the NEP in the context of HPC. We build off the approach from Chapter 3 where we presented a MATLAB implementation of our block version of Krylov-Schur and extend our work to a LAPACK implementation. Here we consider the careful numerical implementation of an extension of Stewart's Krylov-Schur method [64]. We begin with a brief introduction of Krylov decompositions and the essential components of the Krylov-Schur process. Then we outline our block approach and provide a detailed pseudocode of our implementation. Finally we compare our approach to existing LAPACK routines.

Our implementation is able to compute all n eigenvalues of an $n \times n$ matrix. Most implementations of iterative eigensolvers are not able to do so, they compute only partial Schur factorizations. Figure 4.1 shows what happens with MATLAB, for example, when one attempts to compute all n eigenvalues using the function **eigs**. Indeed, some clean-up codes and some mathematical developments are needed in order to handle the last part of the computation. As far as we know, this difficulty is not handled by any iterative eigensolvers. Note that, in general, there is no need for such functionality for very large matrices, the realm of standard iterative eigensolvers, where it would be unreasonable to compute all the eigenvalues of the given matrix. It is however a desirable functionality for our method, so we worked out the details to enable this feature.

```
>> eigs(A,n)
Warning: For nonsymmetric and complex problems,must have number of eigenvalues k < n-1.
 Using eig instead.
> In eigs>checkInputs at 881
  In eigs at 94
```

Figure 4.1: Typically, implementations of iterative eigensolvers are not able to compute n eigenvalues for an $n \times n$ matrix. Here is an example using MATLAB's **eigs** which is based on ARPACK.

4.1 The Krylov-Schur Algorithm

Stewart [64] first suggested the Krylov-Schur algorithm as a computationally attractive alternative to Sorensen's IRAM. The two main issues with the IRAM approach, as detailed in Chapter 3, are the need to preserve the structure of the Arnoldi decomposition given in 3.1 and the complexities associated with deflating converged Ritz vectors. Stewart introduced a general Krylov decomposition to address both of these issues. If $A \in \mathbb{C}^{n\times n}$, a Krylov decomposition of order m is given by

$$AV_m = V_m B_m + v_{m+1} b_{m+1}^H \tag{4.1}$$

where $B_m \in \mathbb{C}^{m\times m}$ and the columns of $V_{m+1} = [V_m, v_{m+1}] \in \mathbb{C}^{n\times(m+1)}$ are independent. The columns of V_{m+1} are called the basis for the decomposition and span the associated space of the decomposition. If the basis is orthonormal, the decomposition is said to be orthonormal. The factor B_m is called the Rayleigh quotient of the Krylov decomposition as the Rayleigh-Ritz procedure extends to decompositions of this form. As Stewart describes, this definition removes all the restrictions imposed on an Arnoldi factorization. The matrix B_m and the vector b_{m+1} are allowed to be arbitrary unlike in an Arnoldi factorization where the Rayleigh quotient must be an upper Hessenberg matrix. Additionally, the basis vectors of a Krylov decomposition are not required to be orthonormal, but we will shortly see that orthonormal vectors are the most desirable option in the context of computing. Stewart offered a connection between these two decompositions. A factorization of the form 4.1 is equivalent, that is related by a sequence of translations and similarities, to an Arnoldi decomposition of the form 3.1. If the Hessenberg term is irreducible, then the Arnoldi decomposition is essentially unique. Stewart's work connecting these two factorizations shows one can work with a less restrictive form, that of the Krylov decomposition, and not lose the Krylov subspace property associated with Arnoldi factorizations.

In particular, any Krylov decomposition can be formulated using orthonormal basis vectors V_{m+1} and the Rayleigh quotient may be reduced to Schur form. This

results in a Krylov-Schur decomposition given by

$$AV_m = V_m S_m + v_{m+1} b_{m+1}^H, \tag{4.2}$$

where the columns of V_{m+1} are orthonormal and S_m the upper triangular factor from the complex Schur form. In matrix notation, we have

$$AV_m = V_{m+1} \widetilde{S}_{m+1}, \tag{4.3}$$

where

$$\widetilde{S}_{m+1} = \begin{bmatrix} S_m \\ b_{m+1}^H \end{bmatrix}.$$

The Krylov-Schur method using this decomposition proceeds in much the same manner as IRAM. The Krylov-Schur method uses an expansion phase to extend the underlying Krylov subspace and then employs a contraction phase to purge unwanted Ritz values. The ease with which the Krylov-Schur method accomplishes the latter is one of the reasons why this approach is so attractive. Before turning to a block variant, we outline the main steps in the basic Krylov-Schur iteration in algorithm 4.1.1 and discuss relevant implementation details.

A practical implementation of the Krylov-Schur method begins with some kind of subspace intialization in the form of a Krylov decomposition. For our approach, based on the work in Chapter 2, we begin with a k_s order Arnoldi decomposition as in equation 2.2, but we note that any subspace initialization that fits the perspective of a Krylov decomposition may be used. Note that subspace initialization with Arnoldi was also used in Chapter 3 for our block extension of **jdqr** and in the original code provided by Sleijpen. Before beginning the Krylov-Schur iteration, the Arnoldi decomposition is reduced to a Krylov-Schur decomposition. The structure of the Rayleigh quotient when reduced to Schur form can be seen in Figure 4.2b. Before the iteration begins, this is a k_s order decomposition. The Krylov-Schur iteration first expands the search subspace in the same manner as the Arnoldi process creating

Algorithm 4.1.1: Krylov-Schur

Input: $A \in \mathbb{C}^{n \times n}$, $v \in \mathbb{C}^n$, and desired number of eigenvalues k_{max}

Result: a partial (or full) Schur form given by $Z_{k_{max}}$ and $\widetilde{S}_{k_{max}}$

Initialize subspace with k_s order Arnoldi decomposition, $AV_{k_s} = V_{k_s+1}\widetilde{H}_{k_s+1}$;

Compute Schur form of H_{k_s+1};

Update first k_s columns of V_{k_s+1} and last row of $\widetilde{H}_{k_s+1}$;

The factorization now has the form $AZ_{k_s} = Z_{k_s+1}\widetilde{S}_{k_s+1}$;

while $k \leq k_{max}$ **do**

    Expand the Krylov-Schur decomposition to $AZ_{k+k_f} = Z_{k+k_f+1}\widetilde{S}_{k+k_f+1}$;

    Re-order the Schur form S_{k+k_f+1};

    Update the first $k + k_f$ columns of Z_{k+k_f+1} and the last row of $\widetilde{S}_{k+k_f+1}$;

    Check convergence;

    if *converged* **then**

        Deflate converged eigenvalue and Schur vector;

        Track number of converged eigenvalues, $k = k + 1$;

        Truncate decomposition and adjust active search subspace dimension;

        $AZ_{k+k_s} = Z_{k+k_s+1}\widetilde{S}_{k+k_s+1}$;

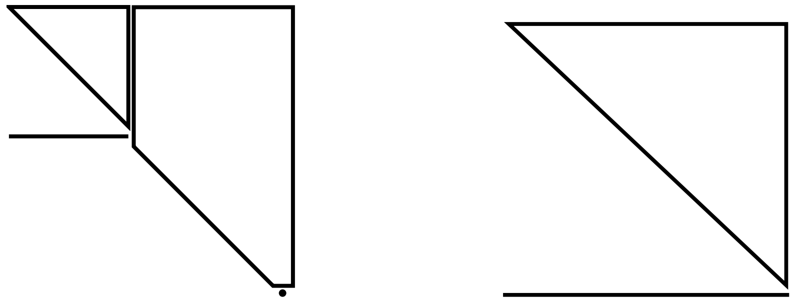
        if $k + k_f = n$ **then**

            Compute Schur form of last projected problem;

            Build Z_n and S_n;

    else

        Truncate decomposition, $AZ_{k+k_s} = Z_{k+k_s+1}\widetilde{S}_{k+k_s+1}$;



(a) Expanded Form (b) Schur Form

Figure 4.2: Structure of the Rayleigh quotient

a k_f order general Krylov decomposition. The structure of the Rayleigh quotient after expansion can be seen in Figure 4.2a. Next the Schur form of the expanded Rayleigh quotient S_{k_f} is computed. Updating the corresponding columns of Z_{k_f} and the bottom row of $\widetilde{S}_{k_f+1}$ changes the structure of the Rayleigh quotient back to its original form as illustrated in Figure 4.2b.

In the next step, the expanded Schur form is re-ordered moving the desired Ritz approximations to the top left of the matrix $\widetilde{S}_{k_f+1}$ and unwanted ones to the bottom right for purging. Different parts of the spectrum may be targeted in the re-ordering phase. The leading component of $b^H_{k_f+1}$, the bottom row of $\widetilde{S}_{k_f+1}$, is then checked against the deflation criteria. If the approximation is not yet acceptable, the expansion is truncated and the iteration continues building off of the truncated Krylov-Schur expansion. Truncation of the Rayleigh quotient is accomplished by selecting the first k_s rows and k_s columns of S_{k_f} and the first k_s components of the bottom row of $\widetilde{S}_{k_f+1}$ to construct $\widetilde{S}_{k_s+1}$. The orthonormal vectors in the first k_s columns of Z_{k_f} and the last column make up the new search subspace Z_{k_s+1}. If the leading component satisfies the convergence criteria 3.18, the converged Ritz pair is deflated.

Deflating converged vectors associated with a single Ritz value is fairly straightforward. After one step of the Krylov-Schur iteration, we have the form

$$A[z_1|\cdots|z_{k_f}] = [z_1|\cdots|z_{k_f+1}] \left[\begin{array}{cc} s & S_{12} \\ 0 & S_{22} \\ \hline b & b_2^H \end{array}\right], \tag{4.4}$$

where s is the component in the first row and first column of S_{k_f} and b is the first component of the vector $b_{k_f+1}^H$. If b satisfies the convergence criterion given in 3.18, it may be set to zero deflating the converged Ritz value s. Stewart suggested a convergence strategy based on having small backward error to ensure backward stability. A similar strategy is used in ARPACK for a converged Ritz value in an Arnoldi decomposition and can be easily extended to a general Krylov decomposition. As Kressner [43] discussed, a more restrictive convergence criterion based on Schur vectors rather than Ritz vectors is possible and we use this strategy. Assuming $k_{con} < k_f$ Ritz values have been deflated, the Krylov decomposition has the form

$$A[Z_{k_{con}}, Z_{k_{con}+k_f}] = [Z_{k_{con}}, Z_{k_{con}+k_f+1}] \left[\begin{array}{cc} S_{k_{con}} & \star \\ 0 & S_{k_f-k_{con}} \\ \hline 0 & b_{k_f-k_{con}}^H \end{array}\right], \tag{4.5}$$

where $S_{k_{con}} \in \mathbb{C}^{k_{con}\times k_{con}}$ is an upper triangular matrix of converged Ritz values. To deflate another Ritz value, the Schur form of $S_{k_f-k_{con}}$ is computed and we have

$$A[Z_{k_{con}}, \widetilde{z}_1, \widetilde{Z}_{k_{con}+k_f-1}] = [Z_{k_{con}}, \widetilde{z}_1, \widetilde{Z}_{k_{con}+k_f}] \left[\begin{array}{ccc} S_{k_{con}} & \star & \star \\ 0 & \lambda & \star \\ 0 & 0 & \widetilde{S}_{k_f-k_{con}-1} \\ \hline 0 & \widetilde{b}_1 & \widetilde{b}_{k_f-k_{con}-1}^H \end{array}\right]. \tag{4.6}$$

We consider the Ritz value λ converged if the corresponding component of the bottom row satisfies the inequality

$$|\widetilde{b}_1| \le \max\{\mathbf{u}\|\widetilde{S}_{k_f-k_{con}-1}\|_F, \mathbf{tol} \times |\lambda|\}, \tag{4.7}$$

where **u** is the unit roundoff and **tol** is a user defined tolerance. If a Ritz value satisfies the criterion 4.7, it satisfies the convergence strategy based on Ritz vectors used by Stewart and ARPACK. If no deflation is possible with λ, the Schur form may be re-ordered and the same test applied to the resulting component of the bottom row. Kressner [43] points out in that this deflation strategy can be considered a variant of AED used in the multishift QR algorithm currently implemented in LAPACK. We will come back to this analogy when we formulate our block approach.

Our implementation of the Krylov-Schur algorithm extends the work of Stewart to compute both partial Schur forms and a complete Schur decomposition. To our knowledge, no current implementation of Krylov-Schur is used to compute a full Schur decomposition. As detailed above, our approach builds an expansion of the form 4.5. In the case of a full Schur form decomposition, the active search subspace is always of size k_s in the contraction phase and of size k_f in the expansion phase. The k_{con} converged Ritz values are locked in the triangular factor $S_{k_{con}}$ and the converged Schur vectors make up the first k_{con} columns of the matrix $Z_{k_{con}+k_f+1}$. When $k_{con}+k_f+1 = n$, the search subspace cannot be extended further and the final projected eigenvalue problem is used to complete the Schur decomposition. The full Schur form is of the form $AZ_n = Z_n S_n$ where $Z_n^H Z_n = I_n$ and S_n is upper triangular.

4.2 The Block Krylov-Schur Algorithm

The extension of algorithm 4.1.1 to a block method is the subject of this section. As we will see, block methods are more complicated than their non-block counterparts and require careful implementation. Block Krylov-Schur expansions are of the form

$$AZ_m = Z_{m+1}\widetilde{S}_{m+1} \tag{4.8}$$

where the $Z_{m+1} = [V_1, \ldots, V_{m+1}]$ has orthonormal columns with each $V_i \in \mathbb{C}^{n\times r}$. The Rayleigh quotient is of the form

$$\widetilde{S}_{m+1} = \begin{bmatrix} S_m \\ b_{m+1}^H \end{bmatrix}$$

where $S_m \in \mathbb{C}^{mr \times mr}$ is upper triangular and $b_{m+1}^H \in \mathbb{C}^{r \times mr}$ is a general matrix. The structure of the Rayleigh quotient $\widetilde{S}_{m+1}$ can be seen in Figure 4.3.

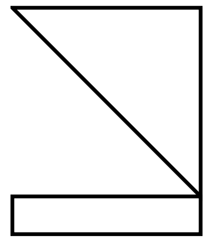


Figure 4.3: Structure of block Krylov-Schur decomposition

Block variants of Krylov-Schur are not new. For symmetric matrices, a block Krylov-Schur approach was suggested by Saad and Zhou [72]. This work addressed several important implementation details. Of note are the handling of the rank deficient case, the orthogonalization scheme, and the incorporation of adaptive block sizes. One complication of blocked methods is that vectors in a new block may become linearly dependent during the iteration. The expansion phase begins with a k_s order block Krylov-Schur decomposition given by

$$AZ_{k_s} = Z_{k_s} S_{k_s} + F b_{k_s+1}^H, \tag{4.9}$$

where $Z_{k_s} \in \mathbb{C}^{n \times k_s r}$ has orthonormal columns, $S_{k_s} \in \mathbb{C}^{k_s r \times k_s r}$ is upper triangular, $F \in \mathbb{C}^{n \times r}$ is such that $F \perp Z_{k_s}$, and $b_{k_s+1} \in \mathbb{C}^{k_s r \times r}$. This is expanded using a block Lanzcos approach, as Saad and Zhou were working with symmetric matrices, to

$$AZ_{k_f} = Z_{k_f} S_{k_f} + F E_{k_f}^H, \tag{4.10}$$

where E_{k_f} is defined as in 2.3. The issue is ensuring that $F \perp Z_{k_f}$. When F is of full rank, the correct augmentation vectors have been located and we may proceed with the block Krylov-Schur iteration. In the case where F is rank deficient, care must be taken. The strategy suggested by Saad and Zhou [72] computes a rank revealing pivoted QR factorization of F. For our discussion, let the rank$(F) = f$ and the

pivoted QR factorization be given by

$$FP = QR$$

where $Q = [Q_f, Q_{r-f}]$. To ensure Q_{r-f} is not in the range of Z_{k_f}, they propose a single vector version of Gram-Schmidt with reorthogonalization and note that this is the same strategy as the DGKS method used in ARPACK [47]. Vectors from Q_{r-f} in $\text{range}\left(Z_{k_f}\right)$ are counted, replaced with random vectors and then orthonormalized against the rest. We mention this to point out that we opt for a superior orthogonalization scheme, at a slight cost. The method formulated in [72] also incorporated adaptive block sizes which is the subject of future work in our endeavors.

A block approach for nonsymmetric matrices is also suggested by Baglama [7] to compute $k < n$ eigenvalues and eigenvectors of an $n \times n$ matrix A. Here the compact WY representation is used in conjunction with an augmented block Arnoldi approach that is restarted using Schur vectors. MATLAB codes are developed to replicate LAPACK codes and the results are compared to ARPACK. Our approach has similar theoretical foundations, but our approaches differ. The major difference between the approaches is that we explicitly form the Krylov-Schur decomposition rather than restarting using Schur vectors. We also use different versions of the compact WY form of the block Householder reflectors as we use existing LAPACK routines. In our MATLAB implementation in Chapter 3, we implemented our own versions of several LAPACK routines for the purposes of comparing our LAPACK implementation. Lastly, we extend the Krylov-Schur foundation to potentially compute a full Schur factorization where the code **abheigs** can only compute a limited number of eigenvalues. As we will see, the shared theoretical foundations lead to algorithms with very different performance. Finally, we work exclusively in LAPACK with an eye on transitioning to the frameworks like PLASMA or MAGMA.

Our approach is an extension of our novel implementation of the Krylov-Schur method in algorithm 3.1.8. We now discuss implementation of the major steps in one

sweep of our approach and provide a detailed pseudocode. As before, we begin with a subspace initialization phase and building off our work in Chapter 2, we start with a k_s order block Arnoldi factorization as in 2.4 where k_s is a multiple of the block size r. Here the number of blocks in the contraction phase is given by b_s so that $k_s = b_s r$, and the number of blocks in the expansion phase is given by b_f so that the size of the expanded search subspace is $k_f = b_f r$. As before, k_{max} is the desired number of eigenvalues we wish to compute and k_{con} is the number of converged eigenvalues.

As outlined earlier, we begin by initializing our search subspace using block Arnoldi. Our block Arnoldi routine uses a variation on the compact WY representation of the Householder reflectors. For our initial decomposition, we use Algorithm 4.2.1 to construct $W \in \mathbb{C}^{n \times k_s + r}$ and $\widetilde{H} \in \mathbb{C}^{(k_s+r) \times k_s}$ such that

$$AW(:, 1 : k_s) = W(:, 1 : k_s + r)\widetilde{H}(1 : k_s + r, 1 : k_s), \tag{4.11}$$

where $W = [V_1, \ldots, V_{b_s}]$ has a compact WY representation and each $V_i \in \mathbb{C}^{n \times r}$. Here xGEQRT is used on blocks of size r creating $T = [T_1, \ldots, T_{b_s}]$ where each $T_i \in \mathbb{C}^{r \times r}$, $i = 1, \ldots, b_s$, corresponds to a block V_i in the search subspace.

Before beginning the Krylov-Schur iteration, we compute the Schur form of the Rayleigh quotient

$$\widetilde{H}(1 : k_s, 1 : k_s)U = US,$$

where $U, S \in \mathbb{C}^{k_s \times k_s}$, $UU^H = I$, and S is upper triangular. Updating the first k_s columns of W and the last r rows of $\widetilde{H}$ we have the initial Krylov-Schur decomposition $AZ = Z\widetilde{S}$, where $Z(:, 1 : k_s) = W(:, 1 : k_s)U$ and

$$\widetilde{S} = \begin{bmatrix} S \\ b^H U \end{bmatrix}.$$

Reduction to Schur form requires a couple computations. First, xGEHRD is used to reduce the band Hessenberg Rayleigh quotient to Hessenberg form and xUNMQR is used to accumulate the corresponding reflectors. Next xHESQR is used to compute

Algorithm 4.2.1: Block Arnoldi subspace initialization

Input: $A \in \mathbb{C}^{n \times n}$, $U \in \mathbb{C}^{n \times r}$, b_s

Result: $AQ = Q\widetilde{H}$ with $Q \in \mathbb{C}^{n \times b_s r}$ and $\widetilde{H} \in \mathbb{C}^{b_s r \times b_s r}$

for $j = 0 : b_s$ **do**

  Compute the QR factorization of $U(jr+1:n, :)$:

  xGEQRT($U[jr]$, T);

  Store the result in V: xLACPY(U, $V[jnr]$);

  The reflectors for the compact WY form of Q are in the lower triangular part of V with associated triangular factors in T;

  Explicitly build Q by accumulating the reflectors in reverse order:

  for $k = j : -1 : 0$ **do**

    xGEMQRT($V[knr + kr]$, $T[krr]$, $Q[jrn + kr]$);

  if $j < b_s$ **then**

    Update U with reflectors from previous columns factorizations:

    for $k = 0 : j$ **do**

      xGEMQRT($V[krn + kr]$, $T[krr]$, $U[kr]$);

the upper triangular factor S and accompanying reflectors. The structure of this

Algorithm 4.2.2: Schur Decomposition of Rayleigh Quotient

Input: $H \in \mathbb{C}^{k \times k}$

Result: Upper triangular S and unitary R such that $HR = RS$

Initialize R with the identity: $R = I_k$;

Compute the Hessenberg reduction: xGEHRD(H, τ);

Apply reflectors to build R: xUNMHR(H, τ, R);

Compute Schur form and build Schur vectors: xHSEQR(H , W, R);

initial step is illustrated in Figure 4.4. After this initialization phase, our block

Algorithm 4.2.3: Block Krylov Schur

Input: $A \in \mathbb{C}^{n\times n}$, r, k_s, k_f and k_{max}

Result: $AZ = ZS$ with $Z \in \mathbb{C}^{n\times k_{max}}$ and $S \in \mathbb{C}^{k_{max}\times k_{max}}$

1 Initialize subspace using Arnoldi as in Algorithm 4.2.1
$AQ(:,1:k_s) = Q(:,1:k_s+r)\widetilde{H}(1:k_s+r,1:k_s)$;

2 Compute Schur form of $\widetilde{H}(1:k_s,1:k_s)$ using Algorithm 4.2.2;

3 The factorization now is a Krylov-Schur decomposition
$AZ(:,1:k_s) = Z(:,1:k_s+r)\widetilde{S}(1:k_s+r,1:k_s)$;

4 $k_{con} = 0$;

5 **while** $k_{con} \leq k_{max}$ **do**

6   Expand the Krylov-Schur decomposition using Algorithm 4.2.4 to
    $AZ = Z\widetilde{S}$ where $Z \in \mathbb{C}^{n\times(k_{con}+k_f+r)}$ and $\widetilde{S} \in \mathbb{C}^{(k_{con}+k_f+r)\times(k_{con}+k_f)}$;

7   Re-order the $k_f \times k_f$ Schur form in the active part of $\widetilde{S}$;

8   Update active parts of Z and $\widetilde{S}$;

9   Check convergence;

10   **if** *converged* **then**

11   Deflate n_{con} converged eigenvalue(s) and Schur vector(s);

12   Adjust active region by locking converged approximations,
        $k_{con} = k_{con} + n_{con}$;

13   Truncate decomposition and adjust active search subspace dimension
        $AZ(:,1:k_{con}+k_s) = Z(:,1:k_{con}+k_s+r)\widetilde{S}(1:k_{con}+k_s+r,1:k_{con}+k_s)$;

14   **else**

15   Truncate decomposition;

16         $AZ(:,1:k_{con}+k_s) = Z(:,1:k_{con}+k_s+r)\widetilde{S}(1:k_{con}+k_s+r,1:k_{con}+k_s)$;

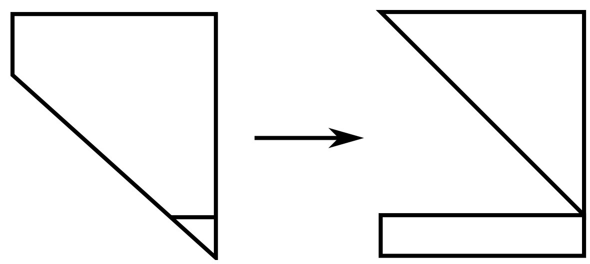


Figure 4.4: Subspace initialization and block Krylov-Schur decomposition

Krylov-Schur approach enters a cycle of expansion and contraction of the search subspace until a Ritz approximation is ready to be deflated. The general outline of the approach is detailed in Algorithm 4.2.3. The expansion phase proceeds in the same manner as our block Arnoldi iteration and with nearly the same structure of Algorithm 4.2.1. Expanding an existing factorization could be accomplished by 4.2.1 if eigenvalues converged r at a time, but this does not seem to be the case based on our numerical experiments. To navigate this slight issue, we use xGEQRT to compute any necessary QR factorizations and store the diagonal of the T factors returned by xGEQRT, as these scalars correspond to the elementary Householder reflectors if we were to proceed by eliminating a column at a time. We then use xLARFT to construct a "big" T for the appropriately sized compact WY form using one vector of all the diagonal entries. If k_{con} eigenvalues have converged, then after expansion the compact WY form used in our approach will be have lower unit triangular $Y \in \mathbb{C}^{n\times(k_{con}+k_f+r)}$ and triangular factor $T \in \mathbb{C}^{(k_{con}+k_f+r)\times(k_{con}+k_f+r)}$. Computing this additional T adds unnecessary flops to our computation. This motivates a possible addition to LAPACK that allows for flexibility in the structure of the T factor.

In a more general context, our approach requires the computation of a QR factorization of an $m \times n$ matrix B. At the conclusion of the calculation, we want the Householder reflectors, $m \times n$ unit lower triangular Y, the $n \times n$ upper triangular R factor, and a specific version of the T factor. The standard output of xGEQRT is a

$b \times n$ matrix T that is a sequence of $b \times b$ triangular factors where b is the block size. If our matrix B is partitioned in two parts as in $B = [B_1, B_2]$ where B_1 is $m \times n_1$ and B_2 is $m \times n_2$ such that $n = n_1 + n_2$, we may desire to compute a compact WY representation in two steps. That is, first compute $R_{1,1} \in \mathbb{C}^{n_1 \times n_1}$, $Y_1 \in \mathbb{C}^{m \times n_1}$ and $T_1 \in \mathbb{C}^{n_1 \times n_1}$ so that $B_1 = (I - Y_1 T_1 Y_1^H) R_{1,1}$. Next we desire to compute the remaining components for the compact WY form of the entire matrix B. We desire to compute $R_{1,2} \in \mathbb{C}^{n_1 \times n_2}$, $R_{2,2} \in \mathbb{C}^{n_2 \times n_2}$, $Y_2 \in \mathbb{C}^{m \times n_2}$, and $T_2 \in \mathbb{C}^{n_2 \times n_2}$ so that the end result

$$\begin{bmatrix} R_{1,1} & R_{1,2} \\ Y_1 & R_{2,2} \\ Y_1 & Y_2 \end{bmatrix} \text{ with } T = [T_1, T_2],$$

is a more general version of the current output of xGEQRT. Here the T factor is a sequence of smaller triangular factors, possibly of various sizes. This fits our application nicely and possibly others. Currently LAPACK does not have this functionality, but the computation of T_2 could be achieved by adjusting xLARFT accordingly. The other components may be computed using existing LAPACK kernels.

In addition to this slight adjustment, the compact WY form of our search subspace may have an additional component. As discussed in our Matlab implementation in Chapter 3, at the completion of one sweep we construct a new compact WY representation of the truncated search subspace. In doing so, we generate an additional diagonal matrix with diagonal elements ± 1 as in 3.7, which we must take into consideration. The details of expanding the Krylov decomposition using existing LAPACK routines may be found in Algorithm 4.2.4. The expanded Rayleigh quotient in the block Krylov decomposition has the structure depicted in Figure 4.5.

Next we compute the Schur form of the active $k_f \times k_f$ part of $\widetilde{S}$. This is again accomplished by Algorithm 4.2.2. The next step is to reorder the Schur form moving the desired Ritz values to the upper left of the active window and the unwanted ones to the bottom right of $\widetilde{S}$ to be purged in the truncation step. This crucial step

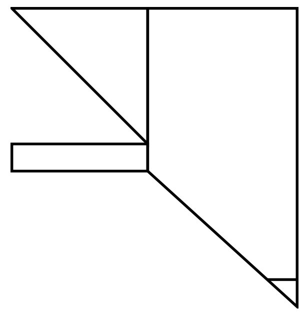


Figure 4.5: Expanded block Krylov decomposition

allows us to compute eigenvalues in desired regions of the spectrum. As mentioned in Chapter 3, we may want to incorporate Kressner's approach to maximize our use of BLAS 3 operations. Rather than calling xTRSEN we employ the same approach used in our MATLAB implementation and reorder the Schur form by hand. Optimizing the performance of this important step is the subject of future work.

For the remainder of this discussion, we will assume k_{con} of the k_{max} desired eigenvalues have converged, with $k_{con} < k_{max}$. The active part of the search subspace is the $k_f + r$ columns of $Z(:, k_{con}+1 : k_{con}+k_f+r)$ and the active part of the Rayleigh quotient is $S(k_{con}+1 : k_{con}+k_f+r, k_{con}+1 : k_{con}+k_f)$. Having reordered the Schur form of the active part of the Rayleigh quotient, we have a factorization of the form

$$AZ(:, 1 : k_{con}+k_f) = Z(:, 1 : k_{con}+k_f+r) \left[\begin{array}{ccc} S_{k_{con}} & \star & \star \\ 0 & \lambda & \star \\ 0 & 0 & \widetilde{S}_{k_f-k_{con}-r} \\ \hline 0 & \widetilde{b}_1 & \widetilde{b}_{k_f-k_{con}-r} \end{array}\right], \tag{4.12}$$

where $\widetilde{b}_1 \in \mathbb{C}^r$ and $\widetilde{b}_{k_f-k_{con}-r} \in \mathbb{C}^{r\times(k_f-k_{con}-r)}$. In the unblocked case ($r = 1$), we needed only to check if the first element in the bottom row satisfied the convergence criteria given in 4.7. It is worth recalling that if 4.7 is satisfied, then the approximation also satisfies the stopping criteria based on Ritz vectors used in ARPACK. Our

Algorithm 4.2.4: Expansion of Krylov decomposition with compact WY form

Input: $U \in \mathbb{C}^{n \times r}$ and Krylov decomposition of size $k_s = b_s r$

Result: Krylov decomposition of size $k_f = b_f r$

for $j = b_s + 1 : b_f$ **do**

  Compute QR factorization of $U(k_{con} + jr + 1 : n, :)$

  xGEQRT($U[k_{con} + jr]$, T);

  Store reflectors in V with H is stored in upper part of V

  xLACPY(U, $V[k_{con}n + jnr]$);

  Accumulate scalars for construction of large T factor

  for $i = 0 : r - 1$ **do**

    $\tau[k_{con} + i + jr] = T[i + ir]$;

  Build full T: xLARFT(V, T);

  Update next part of Krylov sequence

  Explicitly build next block of Z: xGEMQRT(V, T, $Z[k_{con}n + jnr]$);

  if $j < b_f$ **then**

    If needed, compute the new U: xGEMM(A, $Z[k_{con}n + jnr]$, U);

    Update with reflectors: xGEMQRT(V, T, U);

    Finish applying Z with additional piece of compact WY form:

    xGEMM(R, U, U_s); xLACPY(U_s, U);

extension of the convergence criteria in 4.7 to the block case requires

$$\|\widetilde{b}_1\|_2 \leq \max\left\{\mathbf{u}\|\widetilde{S}_{k_f - k_{con} - r}\|_F, tol \times |\lambda|\right\}, \tag{4.13}$$

where $\lambda = S(k_{con} + 1, k_{con} + 1)$, $\mathbf{u}$ is machine precision, and *tol* is a user supplied tolerance. If the approximation satisfies 4.13, the r elements in $\widetilde{b}_1$ are set to zero deflating the Ritz value λ. This is similar to AED in the multishift QR algorithm presented by Braman et al. [16].

If our approximation is satisfactory, we lock the converged eigenvalue and contract the search subspace. Truncation after deflation is detailed in Algorithm 4.2.5. As illustrated in the numerical experiments of Chapter 3, occasionally more than one approximation is ready to be deflated. Our approach checks for multiple deflations each sweep, but as mentioned before, very rarely were r eigenvalues ready to be deflated. If no approximations are acceptable, we purge the unwanted Ritz values by truncating the block Krylov-Schur decomposition. The simplicity of the contraction phase is certainly one of the attractive features of Krylov decompositions, especially compared to Arnoldi decompositions. Again, details may be found in Algorithm 4.2.5. Contraction of a block Krylov-Schur decomposition is illustrated in Figure 4.6 with the Ritz pairs to be purged highlighted in blue.

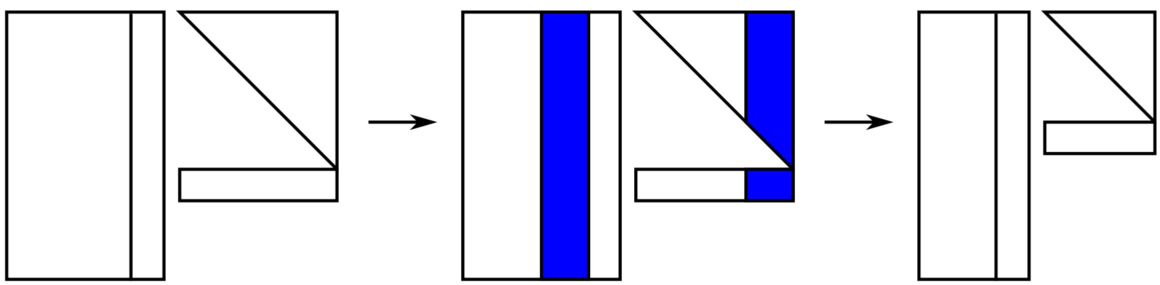


Figure 4.6: Truncation of block Krylov-Schur decomposition

Algorithm 4.2.5: Truncation of block Krylov-Schur decomposition

Input: $H \in \mathbb{C}^{(k_{con}+k_f+r)\times(k_{con}+k_f)}$ and $Z \in \mathbb{C}^{n\times(k_{con}+k_f+r)}$

Result: $H \in \mathbb{C}^{(k_{con}+k_s+r)\times(k_{con}+k_s)}$ and $Z \in \mathbb{C}^{n\times(k_{con}+k_s+r)}$

$n_{con} = 0$;

Check convergence, deflate n_{con} if possible, and truncate;

if $n_{con} > 0$ **then**

  $\lambda[k_{con}] = H[k_{con} + k_{con}k_d]$;

  $k_{con} = k_{con} + 1$;

Restore identity in Q

$Q(k_{con}+k_s+r+1 : n, k_s+k_{con}+r+1 : k_{con}+k_f+r) = \text{eye}(n-k_s-r-k_{con}, k_f-k_s)$;

Copy pieces from Z to Q with xLACPY: $Q(:, 1 : k_{con} + k_s) = Z(:, 1 : k_{con} + k_s)$;

$Q(:, k_{con}+k_s+1 : k_{con}+k_s+r) = Z(:, k_{con}+k_f-n_{con}+1 : k_{con}+k_f-n_{con}+r)$;

Then copy Q back to Z;

Explicitly deflate and contract H using xLACPY:

$H(1 : k_{con} + k_s, 1 : k_{con} + k_s) = triu(H(1 : k_{con} + k_s, 1 : k_{con} + k_s))$;

$H(k_{con} + k_s + 1 : k_{con} + k_s + r, 1 : k_{con} + k_s) = S(k_{con} + k_f - n_{con} + 1 :$

$k_{con} + k_f + r - n_{con}, 1 : k_s + k_{con})]$;

Compute updated compact version of $Z = (I - YTY^H)R$ with

xGEQRT(Q,T);

Store reflectors in V;

4.3 Numerical Results

We present one sample run performed on one node of the colibri cluster at UC Denver. One node has 16 shared memory core. We take $m = 4,000$ for the matrix size, initialize a random (dense) matrix, and compute the 5 largest eigenvalues of A. We use a block size of 20 for the Krylov Schur algorithm with all other parameter set to their default. The results are presented in Figure 4.7. The results for the LAPACK ZGEES reduction to Schur form are presented in Figure 4.8. The parallelism in this experiment is achieved by multithreaded BLAS.

Regarding block Krylov Schur, the algorithm converges in 2,171 BMVPs which corresponds to 65,070 MVPs. We can see nice scalability (on the right) and speedup (on the left). The scalability is much better than for LAPACK's ZGEES. However (1) **bKS** only computes 5 eigenvalues, while LAPACK's ZGEES computes all eigenvalues (4,000), and (2), in sequential, **bKS** is 10x slower than LAPACK. So even though the scalability is much better, the interest is still limited.

The next step for this implementation is to off-load the matrix-vector product (blocked or not) to GPU acceleration. The idea is to set the matrix A on the GPU memory once and for all and use the GPU only for matrix-vector product. So the vectors would move back and forth from CPU to GPU but not the coefficient matrix. Our profiling indicates us that for the experiment presented here in the sequential case 89.20% of the computation time is spent in matrix-vector products. So speeding up the matrix-vector product with GPU could potentially bring a speed up of 10x. We also profiled $m = 8,000$ and $b = 1$ and in this case 99.13% of the computation time is spent in matrix-vector products. So speeding up the matrix-vector product with GPU could potentially bring a speed up of 100x.

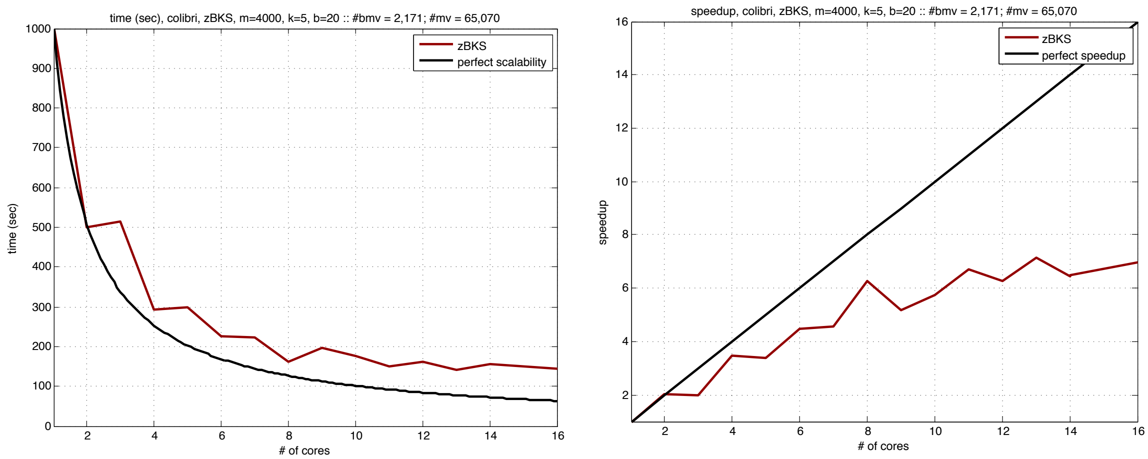


Figure 4.7: Scalability experiments for our block Krylov-Schur algorithm to compute the five largest eigenvalues of a $4,000 \times 4,000$ matrix.

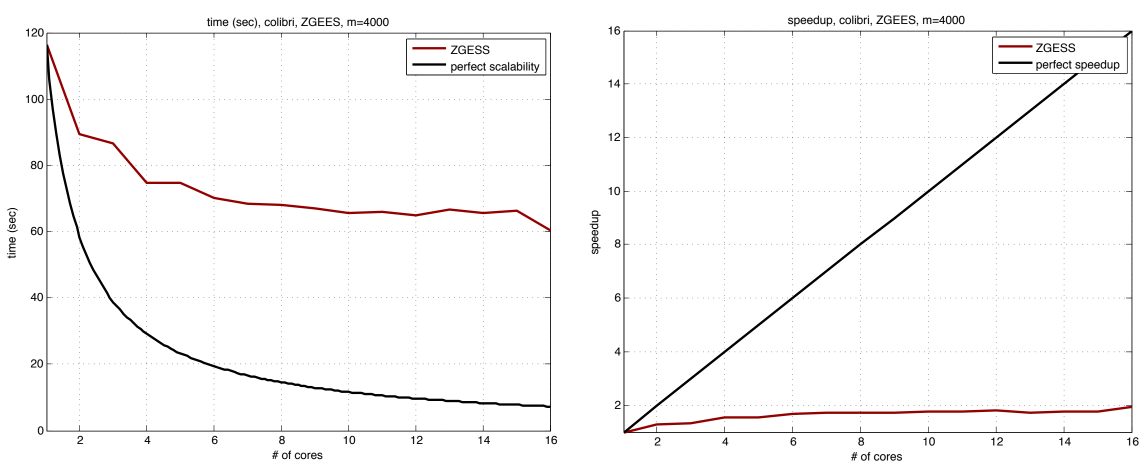


Figure 4.8: Scalability experiments for LAPACK's ZGEES algorithm to compute the full Schur decompositon of a $4,000 \times 4,000$ matrix.

4.4 Conclusion and Future Work

We have implemented block Krylov-Schur in a C code using LAPACK subroutines. The code is robust and supports any block sizes and can compute any number of eigenvalues. The code follows the MATLAB implementation of Chapter 3 but it has not been optimized yet. Future work includes optimization of the code, in particular, we would like to off load the matrix-vector product to GPU units and to incorporate our fast Krylov expansion from Chapter 2. We are also interested in trying our method for parallel distributed (heterogeneous or not) architectures. Additionally, we would like to use sparse matrices when collecting timing results and compare to ARPACK. Regarding dense matrices, we do need to compare to ScaLAPACK as well as of now we only compare to LAPACK with multithreaded BLAS. Finally, we hope to possibly release our code for the scientific computing community.

5. Conclusion

This work endeavored to examine the challenges of the NEP in the context of HPC. To that end, we explored several algorithmic options and available software. Chapter-by-chapter the results are as follows.

In Chapter 2, we presented a discussion on block Arnoldi expansion. We introduced a tile algorithm using Householder reflectors. Our code was implemented within the PLASMA framework which required the augmentation of several existing PLASMA routines to accommodate operations on sub-tiles. We compared two algorithmic variations of our approach achieved by using two different trees (binomial and pipeline). Preliminary results indicate that our Arnoldi implementation performs significantly better than a reference implementation. Future work includes getting more descriptive upper bounds for performance, obtaining a closed form for the critical path and benchmarking our code with sparse matrices.

In Chapter 3, we turned our attention to "iterative" eigensolvers used to compute the partial Schur form of a matrix. We implemented explicitly restarted block Arnoldi with deflation, block Krylov-Schur, and block Jacobi-Davidson, all with Householder orthogonalization and in Matlab. We compared our implementations to publicly available codes for the NEP and results from the literature. Our numerical study showed that our block Krylov-Schur implementation performs comparably to current standards among iterative eigensolvers for sparse matrices.

In Chapter 4, we presented a C code of our block Krylov-Schur implementation using LAPACK subroutines. The code, built off our Matlab implementation of Chapter 3, is robust and can compute any number of desired eigenvalues. The implementation achieves good scalability, but a majority of the computation time is spent in the matrix-vector product. Future work includes optimization of the code and off-loading the matrix-vector products to GPUs.

REFERENCES

[1] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, and Hatem Ltaief. Plasma users' guide. Technical report, ICL UTK, 2009.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] Peter Arbenz, Martin Bečka, Roman Geus, Ulrich Hetmaniuk, and Tiziano Mengotti. On a parallel multilevel preconditioned Maxwell eigensolver. *Parallel Comput.*, 32(2):157–165, 2006.

[4] W. E. Arnoldi. The principle of minimized iteration in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.*, 9:17–29, 1951.

[5] Marc Baboulin, Jack Dongarra, and Stanimire Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. Technical Report 200, LAPACK Working Note, 2004.

[6] J. Baglama, D. Calvetti, and L. Reichel. Algorithm 827: irbleigs: a MATLAB program for computing a few eigenpairs of a large sparse Hermitian matrix. *ACM Trans. Math. Software*, 29(3):337–348, 2003.

[7] James Baglama. Augmented block Householder Arnoldi method. *Linear Algebra Appl.*, 429(10):2315–2334, 2008.

[8] Z. Bai and J. Demmel. On a block implementation of Hessenberg multishift QR iteration. Technical Report 8, LAPACK Working Note, 1989.

[9] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Trans. Math. Softw.*, 36(3):13:1–13:23, July 2009.

[10] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[11] BLAS. Basic linear algebra subprograms. http://www.netlib.org/blas/.

[12] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix market: a web resource for test matrix collections. In *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software: assessment and enhancement*, pages 125–137, London, UK, UK, 1997. Chapman & Hall, Ltd.

[13] Henricus M. Bouwmeester. *Tile Algorithms For Matrix Computations On Multicore Architectures*. PhD thesis, University of Colorado Denver, 2012.

[14] Karen Braman. Middle deflations in the QR algorithm. Householder Symposium XVII, 2008.

[15] Karen Braman, Ralph Byers, and Roy Mathias. The multishift QR algorithm. I. Maintaining well-focused shifts and level 3 performance. *SIAM J. Matrix Anal. Appl.*, 23(4):929–947 (electronic), 2002.

[16] Karen Braman, Ralph Byers, and Roy Mathias. The multishift QR algorithm. II. Aggressive early deflation. *SIAM J. Matrix Anal. Appl.*, 23(4):948–973 (electronic), 2002.

[17] Jan Brandts. The Riccati algorithm for eigenvalues and invariant subspaces of matrices with inexpensive action. *Linear Algebra Appl.*, 358:335–365, 2003. Special issue on accurate solution of eigenvalue problems (Hagen, 2000).

[18] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, September 2008.

[19] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35:38–53, 2009.

[20] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the gram-schmidt qr factorization. *Mathematics of Computation*, 30(136):pp. 772–795, 1976.

[21] Ernest R. Davidson. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *J. Computational Phys.*, 17:87–94, 1975.

[22] Timothy A. Davis. The university of florida sparse matrix collection. *NA DIGEST*, 92, 1994.

[23] J. Dongarra, S. Hammarling, and D. Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. Technical Report 2, LAPACK Working Note, 1987.

[24] Diederik R. Fokkema, Gerard L. G. Sleijpen, and Henk A. Van der Vorst. Jacobi-Davidson style QR and QZ algorithms for the reduction of matrix pencils. *SIAM J. Sci. Comput.*, 20(1):94–125 (electronic), 1998.

[25] J. G. F. Francis. The QR transformation: a unitary analogue to the LR transformation. I. *Comput. J.*, 4:265–271, 1961/1962.

[26] J. G. F. Francis. The QR transformation. II. *Comput. J.*, 4:332–345, 1961/1962.

[27] R. Geus. *The Jacobi-Davidson algorithm for solving large sparse symmetric eigenvalue problems with application to the design of accelerator cavities.* PhD thesis, ETH Zurich, 2002.

[28] Gene H. Golub and Charles F. Van Loan. *Matrix Computation.* The Johns Hopkins University Press, 1996.

[29] R. Granat, B. Kågström, and D. Kressner. A novel parallel QR algorithm for hybrid distributed memory HPC systems. Technical Report 216, LAPACK Working Note, 2004.

[30] G. Henry, D. Watkins, and J. Dongarra. A parallel implementation of the nonsymmetric QR algorithm for distributed memory architectures. LAPACK Working Note 121, UT-CS, 1997. UT-CS-97-352, March 1997.

[31] Greg Henry and Robert van de Geijn. Parallelizing the QR algorithm for the unsymmetric algebraic eigenvalue problem: Myths and reality. Technical Report 79, LAPACK Working Note, 1994.

[32] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, 2002.

[33] M.E. Hochstenbach. Jacobi-Davidson Gateway. http://www.win.tue.nl/casa/research/topics/jd/.

[34] Michiel E. Hochstenbach and Yvan Notay. Controlling inner iterations in the Jacobi-Davidson method. *SIAM J. Matrix Anal. Appl.*, 31(2):460–477, 2009.

[35] Roger A. Horn and Charles R. Johnson. *Matrix Analysis.* Cambridge University Press, 1985.

[36] Gary W. Howell and Nadia Diaa. Algorithm 841: Bhess: Gaussian reduction to a similar banded hessenberg form. *ACM Trans. Math. Softw.*, 31(1):166–185, March 2005.

[37] C.G.J. Jacobi. Über eine neue Auflösungsart der bei der Methode der kleinsten Quadrate vorkommende linearen Gleichungen. *Astronom. Nachr.*, pages 297–306, 1845.

[38] Zhongxiao Jia. A refined iterative algorithm based on the block Arnoldi process for large unsymmetric eigenproblems. *Linear Algebra Appl.*, 270:171–189, 1998.

[39] Lars Karlsson and Daniel Kressner. Optimally packed chains of bulges in multishift QR algorithms. Technical Report 271, LAPACK Working Note, 2012.

[40] Andrew V. Knyazev. Toward the optimal preconditioned eigensolver: locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput.*, 23(2):517–541 (electronic), 2001. Copper Mountain Conference (2000).

[41] Andrew V. Knyazev. Hard and soft locking in iterative methods for symmetric eigenvalue problems. Copper Mountain Conference on Iterative methods, 2004.

[42] D. Kressner. Block algorithms for reordering standard and generalized Schur forms lapack working note 171. LAPACK Working Note 171, UT-CS, 2006. UT-CS-97-352, March 1997.

[43] Daniel Kressner. *Numerical Methods for General and Structured Eigenvalue Problems*. Springer, 2000.

[44] V.N. Kublanovskaya. On some algorithms for the solution of the complete eigenvalue problem. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 1(4):555–570, 1961.

[45] Julien Langou. *Solving large linear systems with multiple right-hand sides*. PhD thesis, L'Institut National Des Sciences Appliquées De Toulouse, 2003.

[46] R. Lehoucq and K. Maschhoff. Implementation of an implicitly restarted block Arnoldi method. *Preprint MCS-P649-0297, Argonne National Lab*, 1997.

[47] R. B. Lehoucq, D. C. Sorensen, and C. Yang. ARPACK user's guide: Solution of large scale eigenvalue problems by implicitly restarted Arnoldi methods., 1997.

[48] Hatem Ltaief, Jakub Kurzak, and Jack Dongarra. Parallel two-stage Hessenberg reduction using tile algorithms for multicore architectures. Technical Report 208, LAPACK Working Note, 2004.

[49] J. Möller. Implementations of the implicitly restarted block Arnoldi method. Technical report, Royal Institute of Technology (KTH), Dept. of Numerical Analysis and Computer Science, 2004.

[50] Ronald B. Morgan. On restarting the Arnoldi method for large nonsymmetric eigenvalue problems. *Math. Comp.*, 65(215):1213–1230, 1996.

[51] Ronald B. Morgan. Restarted block-GMRES with deflation of eigenvalues. *Appl. Numer. Math.*, 54(2):222–236, 2005.

[52] Margreet Nool and Auke van der Ploeg. A parallel Jacobi-Davidson-type method for solving large generalized eigenvalue problems in magnetohydrodynamics. *SIAM J. Sci. Comput.*, 22(1):95–112 (electronic), 2000.

[53] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3), 2009.

[54] Gregorio Quintana-Ortí and Robert van de Geijn. Improving the performance of reduction to Hessenberg form. *ACM Trans. Math. Software*, 32(2):180–194, 2006.

[55] Yousef Saad. *Numerical Methods for Large Eigenvalue Problems*. SIAM, 2011.

[56] Robert Schreiber and Charles Van Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM J. Sci. Statist. Comput.*, 10(1):53–57, 1989.

[57] Jennifer A. Scott. An Arnoldi code for computing selected eigenvalues of sparse, real, unsymmetric matrices. *ACM Trans. Math. Software*, 21(4):432–475, 1995.

[58] Gerard L. G. Sleijpen and Henk A. Van der Vorst. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 17(2):401–425, 1996.

[59] D. C. Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM J. Matrix Anal. Appl.*, 13(1):357–385, 1992.

[60] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM Journal on Scientific Computing*, 23(6):2165–2182, 2002.

[61] Andreas Stathopoulos and James R. McCombs. Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. II. Seeking many eigenvalues. *SIAM J. Sci. Comput.*, 29(5):2162–2188 (electronic), 2007.

[62] G. W. Stewart. A parallel implementation of the QR-algorithm. In *Proceedings of the international conference on vector and parallel computing—issues in applied research and development (Loen, 1986)*, volume 5, pages 187–196, 1987.

[63] G. W. Stewart. *Matrix Algorithms Volume II: Eigensystems*. SIAM, 2001.

[64] G. W. Stewart. A Krylov-Schur algorithm for large eigenproblems. *SIAM J. Matrix Anal. Appl.*, 23(3):601–614 (electronic), 2001/02.

[65] S. Tomov, J. Dongarra, V. Volkov, and J. Demmel. Magma library, version 0.1. http://icl.cs.utk.edu/magma, 08/2009.

[66] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra*. SIAM, 1997.

[67] Homer F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Statist. Comput.*, 9(1):152–163, 1988.

[68] Shunxu Wang. A parallel refined Jacobi-Davidson method for quadratic eigenvalue problems. In *Parallel Architectures, Algorithms and Programming (PAAP), 2010 Third International Symposium on*, pages 111–115, 2010.

[69] David S. Watkins. The transmission of shifts and shift blurring in the QR algorithm, 1992.

[70] David S. Watkins. *The Matrix Eigenvalue Problem.* SIAM, 2007.

[71] Bai Zhaojun, Demmel James, Dongarra Jack, Ruhe Axel, and van der Vorst Henk, editors. *Templates for the Solution of Algebraic Eigenvalue Problems.* Society for Industrial and Applied Mathematics, 2000.

[72] Yunkai Zhou and Yousef Saad. Block Krylov-Schur method for large symmetric eigenvalue problems. *Numer. Algorithms*, 47(4):341–359, 2008.